



**Calhoun: The NPS Institutional Archive**  
**DSpace Repository**

---

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

---

1990-12

# Discrete cosine transform implementation in VHDL

Hu, Ta-Hsiang.

Monterey, California: Naval Postgraduate School

---

<http://hdl.handle.net/10945/27602>

---

Copyright is reserved by the copyright owner

*Downloaded from NPS Archive: Calhoun*



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

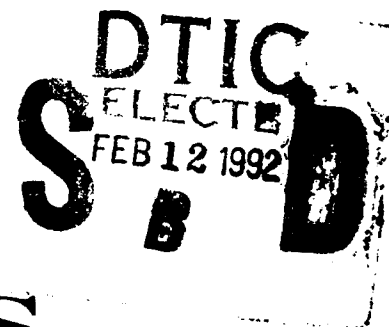
<http://www.nps.edu/library>

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California

2

AD-A245 791



# THESIS

DISCRETE COSINE TRANSFORM IMPLEMENTATION  
IN VHDL

by

Ta-Hsiang Hu

December 1990

Thesis Advisor:  
Thesis Co-Advisor:

Chin-Hwa Lee  
Chyan Yang

Approved for public release; distribution is unlimited.

92-03291



Unclassified

Security Classification of this page

**REPORT DOCUMENTATION PAGE**

1a Report Security Classification <b>Unclassified</b>				1b Restrictive Markings			
2a Security Classification Authority				3 Distribution Availability of Report			
2b Declassification/Downgrading Schedule				Approved for public release; distribution is unlimited.			
4 Performing Organization Report Number(s)				5 Monitoring Organization Report Number(s)			
6a Name of Performing Organization Naval Postgraduate School			6b Office Symbol 62	7a Name of Monitoring Organization Naval Postgraduate School			
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000				7b Address (city, state, and ZIP code) Monterey, CA 93943-5000			
8a Name of Funding/Sponsoring Organization			8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number			
8c Address (city, state, and ZIP code)				10 Source of Funding Numbers			
				Program Element Number	Project No	Task No	Work Unit Accession No
11 Title (Include Security Classification) <b>DISCRETE COSINE TRANSFORM IMPLEMENTATION IN VHDL</b>							
12 Personal Author(s) <b>Ta-Hsiang Hu</b>							
13a Type of Report Master's Thesis		13b Time Covered From To		14 Date of Report (year, month, day) December 1990		15 Page Count 166	
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.							
17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)				
Field	Group	Subgroup	FFT SYSTEM, DCT SYSTEM IMPLEMENTATION				
19 Abstract (continue on reverse if necessary and identify by block number) Several different hardware structures for Fast Fourier Transform(FFT) are discussed in this thesis. VHDL was used in providing a simulation. Various costs and performance comparisons of different FFT structures are revealed. The FFT system leads to a design of Discrete Cosine Transform(DCT). VHDL allows the hierarchical description of a system in structural and behavioral description. In the structural description, a component is described in terms of an interconnection of more primitive components. However, in the behavioral description, a component is described by defining its input/output response in terms of a procedure. In this thesis, the lowest hierarchy level is chip-level. In modeling of the floating point unit AMD29325 behavior, several basic functions or procedures are involved. A number of AMD29325 chips were used in the different structures of the FFT butterfly. The full pipeline structure of the FFT butterfly, controller, and address sequence generator are simulated in VHDL. Finally, two methods of implementation of the DCT system are discussed.							
20 Distribution/Availability of Abstract <input checked="" type="checkbox"/> unclassified/unlimited <input type="checkbox"/> same as report <input type="checkbox"/> DTIC users				21 Abstract Security Classification <b>Unclassified</b>			
22a Name of Responsible Individual Chin-Hwa Lee				22b Telephone (Include Area code) (408) 655-0242		22c Office Symbol EC / Le	

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted

security classification of this page

All other editions are obsolete

Unclassified

Approved for public release; distribution is unlimited.

**Discrete Cosine Transform Implementation In VHDL**

by

**Ta-Hsiang Hu**  
**Captain, Republic of China Army**  
**B.S., Chung-Cheng Institute Of Technology, 1984**

Submitted in partial fulfillment of the requirements  
for the degree of

**MASTER OF SCIENCE IN ELECTRICAL  
ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL**  
**December 1990**

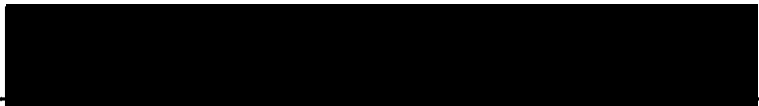
Author:

  
**Ta-Hsiang Hu**

Approved by:

  
**Chin-Hwa Lee, Thesis Advisor**

  
**Chyan Yang, Co-Advisor**

  
**Michael A. Morgan, Chairman**  
**Department of Electrical and Computer Engineering**

## ABSTRACT

Several different hardware structures for Fast Fourier Transform(FFT) are discussed in this thesis. VHDL was used in providing a simulation. Various costs and performance comparisons of different FFT structures are revealed. The FFT system leads to a design of Discrete Cosine Transform(DCT). VHDL allows the hierarchical description of a system in structural and behavioral description. In the structural description, a component is described in terms of an interconnection of more primitive components. However, in the behavioral domain, a component is described by defining its input/output response in terms of a procedure. In this thesis, the lowest hierarchy level is chip-level. In modeling of the floating point unit AMD29325 behavior, several basic functions or procedures are involved. A number of AMD29325 chips were used in the different structures of the FFT butterfly. The full pipeline structure of the FFT butterfly, controller, and address sequence generator are simulated in VHDL. Finally, two methods of implementation of the DCT system are discussed.

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## TABLE OF CONTENTS

I.	INTRODUCTION . . . . .	1
A.	VHDL HARDWARE DESCRIPTION LANGUAGE . . . . .	1
B.	OVERVIEW OF THE THESIS . . . . .	2
II.	FLOATING POINT UNIT . . . . .	5
A.	OVERVIEW OF THE IEEE FLOATING POINT STANDARD FORMAT . . . . .	5
B.	INTRODUCTION TO FLOATING POINT UNIT CHIP AMD29325 . . . . .	9
C.	BASIC MODELING FUNCTIONS OF AMD29325 . . . . .	10
1.	THE ELEMENT FUNCTIONS ASSOCIATED WITH THE ARITHMETICAL OPERATION OF AMD29325 . . . . .	10
2.	THE TOP FUNCTIONS ASSOCIATED WITH THE ARITHMETICAL OPERATIONS OF AMD29325 . . . . .	14
a.	Addition Operation Function . . . . .	14
b.	Subtraction Operation Function . . . . .	15
c.	Multiplication Operation Function . . . . .	15
d.	Division Operation Function . . . . .	15
3.	BEHAVIORAL DESCRIPTION OF THE AMD29325 CHIP	16
III.	THE DATA FLOW DESIGN OF THE FAST FOURIER TRANSFORM	21
A.	OVERVIEW OF THE FAST FOURIER TRANSFORM . . . . .	21
1.	DECIMATION IN TIME(DIT) . . . . .	21

2.	DECIMATION IN FREQUENCY(DIF)	22
B.	COMPARISON OF SEVERAL DATA FLOW CONFIGURATIONS	
	OF THE FAST FOURIER TRANSFORM	24
1.	STRUCTURE 1 OF DIF BUTTERFLY	26
2.	STRUCTURE 2 OF DIF BUTTERFLY	32
3.	STRUCTURE 3 OF DIF BUTTERFLY	33
4.	STRUCTURE 4 OF DIF BUTTERFLY	40
5.	STRUCTURE 5 OF DIF BUTTERFLY	43
6.	STRUCTURE 6 OF DIF BUTTERFLY	46
C.	SOME VHDL BEHAVIORAL MODELS	50
1.	FULL PIPELINE DIF BUTTERFLY STRUCTURE	50
2.	CONTROLLER FOR THE BUTTERFLY STRUCTURE	51
3.	ADDRESS SEQUENCE GENERATOR	52
4.	RAM	59
D.	SIMULATION OF THE DATA FLOW DESIGN OF FFT	60
IV.	THE DATA FLOW DESIGN OF THE DISCRETE COSINE	
	TRANSFORM	68
A.	INTRODUCTION TO DISCRETE COSINE TRANSFORM(DCT)	68
B.	THE DISCRETE COSINE TRANSFORM SYSTEM	
	IMPLEMENTATION	70
V.	CONCLUSION	76
A.	CONCLUSION	76
B.	IMPROVEMENTS AND FUTURE RESEARCH	77
1.	TO IMPLEMENT THREE ADDITIONAL PRECISION	
	FORMATS TO IMPROVE THE ARITHMETIC ACCURACY.	78

2. TO ADD SEVERAL OTHER FUNCTIONS ASSOCIATED WITH THE AMD29325 OPERATION . . . . .	78
3. TO PERFORM THE RADIX 4 FAST FOURIER TRANSFORM IN DIT OR DIF ALGORITHMS. . . . .	78
4. TO IMPROVE THE ADDRESSING SEQUENCE GENERATOR TO REDUCE FETCHING IDENTICAL WEIGHT FACTORS. . . . .	81
5. TO BUILD THE FAST FOURIER TRANSFORM USING SPECIAL "COMPLEX VECTOR PROCESSOR (CVP)" CHIP. . . . .	81
APPENDIX A: THE ELEMENT FUNCTIONS OF THE FPU . . . . .	82
APPENDIX B: THE TOP FUNCTIONS AND BEHAVIOR OF THE FPU . . . . .	92
A. THE TOP FUNCTIONS OF THE FPU . . . . .	92
B. THE BEHAVIOR FUNCTIONS OF THE FPU . . . . .	103
APPENDIX C: THE SOURCE FILE OF THE FPU CHIP AMD29325 . . . . .	105
APPENDIX D: THE SIMPLIFIED I/O PORT OF THE FPU CHIP AMD29325 . . . . .	107
APPENDIX E: THE PIPELINE STRUCTURE OF THE FFT BUTTERFLY . . . . .	109
APPENDIX F: THE ADDRESS SEQUENCE GENERATOR AND CONTROLLER . . . . .	116
APPENDIX G: THE BEHAVIOR OF RAM . . . . .	126
APPENDIX H: THE SOURCE FILE OF THE FFT SYSTEM . . . . .	130
APPENDIX I: THE ACCESSORY FILES . . . . .	148
A. THE SOURCE FILE ASSOCIATED WITH DATA READ . . . . .	148
B. THE SOURCE FILE OF THE CONVERSION BETWEEN FP_NUMBER AND IEEE FORMAT . . . . .	150



LIST OF REFERENCES . . . . .	152
INITIAL DISTRIBUTION LIST . . . . .	153

## LIST OF TABLES

TABLE 3.1	Time space diagram of DIF structure 1.	. .	30
TABLE 3.2	Time space diagram of DIF structure 2.	. .	35
TABLE 3.3	Time space diagram of DIF structure 3.	. .	39
TABLE 3.4	Time space diagram of DIF structure 4.	. .	42
TABLE 3.5	Time space diagram of DIF structure 5.	. .	45
TABLE 3.6	Time space diagram of DIF structure 6.	. .	48
TABLE 3.7	Comparison of 6 DIF butterfly structures.	.	49
TABLE 3.8	Comparison of the FFT result using the MATLAB function and this simulated FFT system.	. . . . .	65
TABLE 5.1	The comparison of total number of arithmetic operations needed in Radix 2 and Radix 4	. . . . .	80

## LIST OF FIGURES

FIGURE 1.1	The designed tree of this thesis . . . . .	4
FIGURE 2.1	The IEEE single precision floating point format . . . . .	6
FIGURE 2.2	Format parameter for the IEEE 754 floating point standard . . . . .	7
FIGURE 2.3	AMD29325 block diagram (adapted from AMD data book) . . . . .	11
FIGURE 2.4	AMD29325 pin diagram (adapted from the AMD data book) . . . . .	12
FIGURE 2.5	AMD29325 operation select (adapted from AMD data book) . . . . .	13
FIGURE 2.6	The entity of a FULL_ADDER . . . . .	16
FIGURE 2.7	Three constructs in VHDL language (adopted from [Ref. 4]) . . . . .	17
FIGURE 2.8	AMD29325 pin description (adapted from the AMD data book) . . . . .	18
FIGURE 3.1	Signal flow graph and the shorthand representation of DIT butterfly . . . . .	22
FIGURE 3.2	8 point FFT using DIT butterfly . . . . .	23
FIGURE 3.3	Signal flow graph and shorthand representation in DIF butterfly . . . . .	24
FIGURE 3.4	8 point FFT using DIF butterfly . . . . .	25

FIGURE 3.5	8 point FFT with DIF butterfly in non-bit-reversal algorithm . . . . .	26
FIGURE 3.6	Two different basic butterflies and their arithmetic operations . . . . .	27
FIGURE 3.7	Butterfly implementation in pipeline structure . . . . .	29
FIGURE 3.8	Butterfly implementation in structure 2 . .	34
FIGURE 3.9	Butterfly implementation in structure 3 . .	38
FIGURE 3.10	Butterfly implementation in structure 4 .	41
FIGURE 3.11	Butterfly implementation in structure 4 .	44
FIGURE 3.12	Butterfly implementation in structure 6 .	47
FIGURE 3.13	Controller flow chart and its logical symbol	53
FIGURE 3.14	The block diagram of address sequence generator and controller . . . . .	55
FIGURE 3.15	Address sequence generator flow chart . .	56
FIGURE 3.16	Timing of read cycle and write cycle (adopted from National CMOS RAM data book) . . . .	61
FIGURE 3.17	The original data flow system of FFT . . .	62
FIGURE 3.18	The revised data flow system of FFT . . .	66
FIGURE 3.19	The flow chart of the universal controller . . . . .	67
FIGURE 4.1	Full pipeline structure to implement the DCT system, the input data come from the FFT system output. . . . .	73
FIGURE 4.2	Block diagram of the universal controller and the FFT system. . . . .	74

FIGURE 4.3	Modified flow chart of the universal controller . . . . .	75
FIGURE 5.1	Butterfly in Radix 4, top is the DIT algorithm, bottom is the DIF algorithm. . . . .	79

## ACKNOWLEDGMENTS

I wish to express my thanks to my Thesis Advisor, Prof. Chin-Hwa Lee, for his understanding, infinite patience, and guidance. I would like to thank my Thesis Co-Advisor, Prof. Chyan Yang who provided a lot of help and enthusiasm when it was greatly needed. Finally, I would like to extend deep appreciation to my family and friends who gave their total support throughout the entire project.

## I. INTRODUCTION

### A. VHDL HARDWARE DESCRIPTION LANGUAGE

VHDL stands for VHSIC Hardware Description Language. "It is a new hardware description language developed and standardized by the U.S. Department of Defense for documentation and specification of CAD microelectronics design" [Ref. 1]. "The language was developed to address a number of recurrent problems in the design cycles, exchange of design information, and documentation of digital hardware. VHDL is technology independent and is not tied to a particular simulator or logic value set. Also it does not force a design methodology on a designer" [Ref. 2]. Many existing hardware description languages can operate at the logic and gate level. Consequently, they are low-level logic design simulators. While VHDL is perfectly suited to this level of description, it can be extended beyond this to higher behavioral levels. For example, it can extend from the level of gate, register, chip, up to the desired system level. VHDL allows hierarchy implementation in two domains, structural and behavioral domains, by digital designers [Ref. 3]. In the structural domain, a component is described in terms of an interconnection of more primitive components. However, in the behavioral domain, a component is

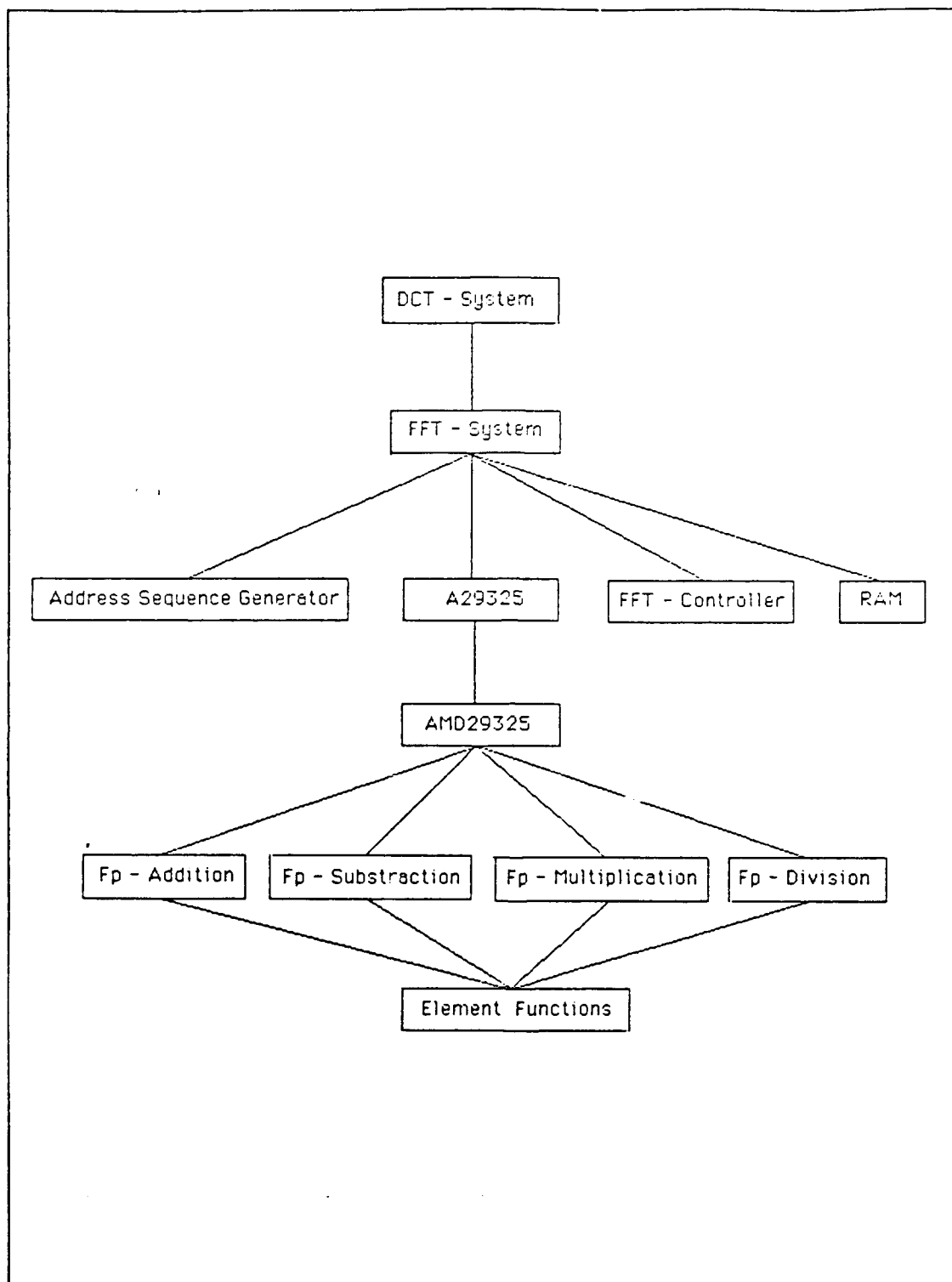
described by defining the input/output response in terms of a procedure. In this thesis, the lowest hierarchy level is at the chip-level. Modeling the behavior at the chip-level is the first task. Then, various structures of FFT system are designed using these primitives, i.e. chips. In order to model these chips accurately Time-delay and hold-up-time as VHDL generic are introduced. Different structures were studied here to compare system performance and costs. The structural modeling and behavioral modeling in VHDL are the main subjects in this thesis. In other words, VHDL is the main language tool to allow for capturing and verifying all the design details. In this thesis, VHDL was used to model at the chip level, a floating point unit, a Discrete Fourier Transform system, and a Discrete Cosine Transform system.

## **B. OVERVIEW OF THE THESIS**

This thesis is divided into five chapters. Chapter I gives a general introduction. Several element functions, four basic operations of the floating point unit AMD29325, and a simplified version A29325 are created in Chapter II. Chapter III includes the designs of the butterfly of a Fast Fourier Transform(FFT) in DIF algorithm, six different kinds of data flow configurations, VHDL RAM models, controller, address sequence generator, and integrated models of the FFT system. Furthermore, in Chapter IV a Discrete Cosine Transform(DCT) is implemented based on the extension of the universal controller



of the FFT system. Finally, Chapter V gives the conclusions and suggestions of possible future research. The hierarchy of the design units created in this thesis can be summarized in a tree shown in Figure 1.1. The efforts start at the bottom of the tree, and end at the top. Various nodes in the tree will be explained in detail in the following chapters.



**FIGURE 1.1** The design tree of this thesis.

## II. FLOATING POINT UNIT

### A. OVERVIEW OF THE IEEE FLOATING POINT STANDARD FORMAT

Sometimes applications require numbers with large numerical range that can not be stored as integers. In these situations, there may also be a need to represent NAN( not a number ) or infinite number. Fixed point number representation is not sufficient to support these needs. In this situation, a floating point number is used. There are several formats for representing floating point numbers.

Any floating point format usually includes three parts, a sign bit, an exponential bit pattern, and a mantissa bit pattern. Different computer systems such as CDC 7600, DEC, VAXII, HONEYWELL 8200, IBM 3303 might use different floating point formats. The variations occur in the number of bits allocated for the exponent and mantissa patterns, how rounding is carried out, and the actions taken while underflow and overflow occur. Therefore, there is a need for a standard floating point format to allow the interchange of floating point data easily.

Usually, the value of a floating point format is

$$(\text{sign})\text{Mantissa} * 2^{\text{exponent}} \quad (2.1)$$

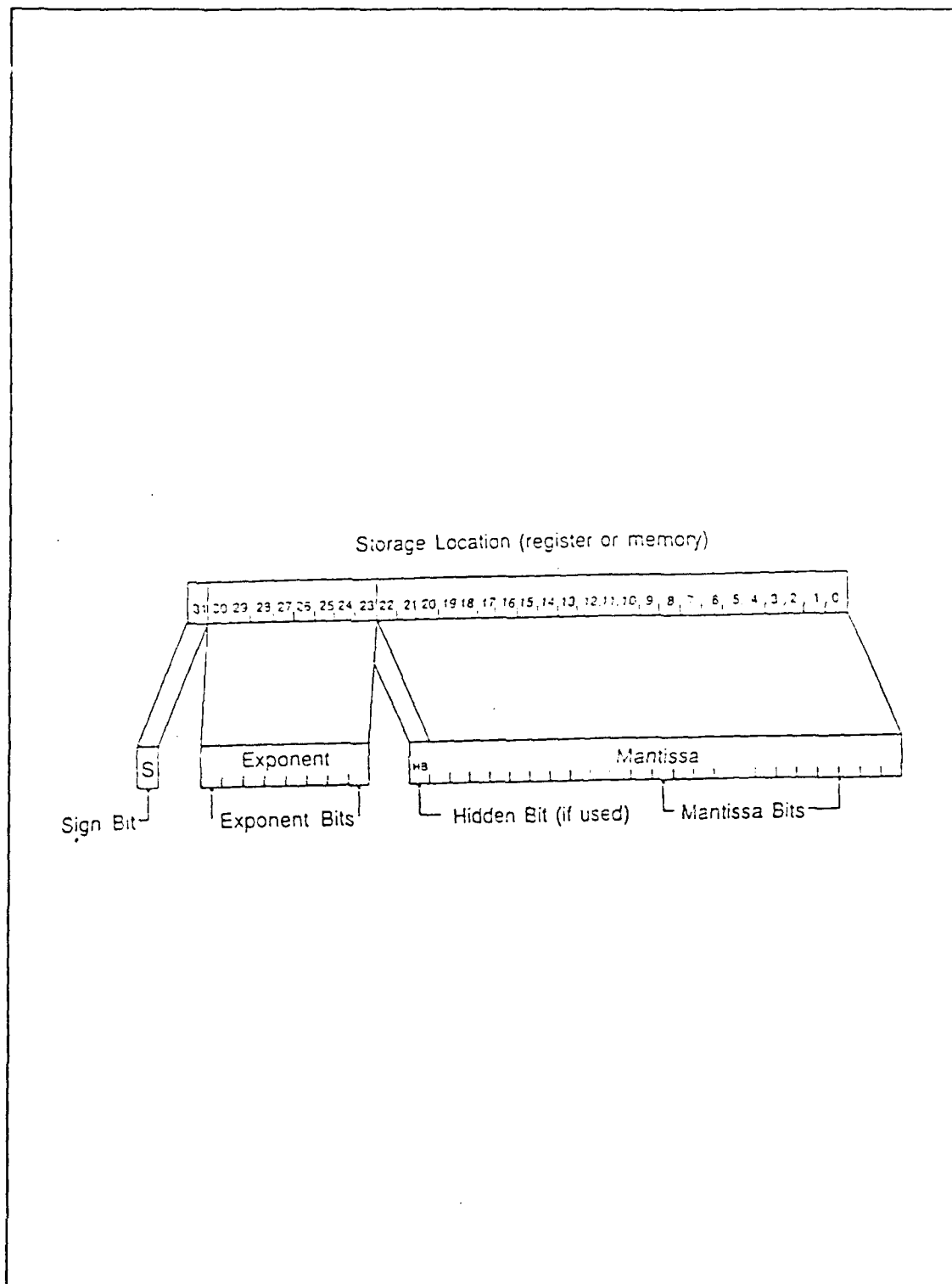


FIGURE 2.1 The IEEE single precision floating point format.

In Figure 2.1 [Ref. 4], the IEEE single precision floating point format is shown with the sign bit, exponent bits and mantissa bits. The IEEE single precision floating format contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. There is an important fact that 1 bit is hidden in the mantissa. Consequently, the actual size of fraction is 24. In other words, the actual number of bits of the fraction is that of the mantissa value, from the 22th bit down to the zero bit in Figure 2.1, added by 1. In this case the actual value of the fraction is

$$1.0 < \text{actual fraction} < 2.0 \quad (2.2)$$

The IEEE floating point format supports not only single precision but also other precision formats. The other precision formats are shown in Figure 2.2 [Ref. 5].

	Single	Single extended	Double	Double extended
$p$ (bits of precision)	24	$\geq 32$	53	$\geq 64$
$E_{\max}$	127	$\geq 1023$	1023	$\geq 16383$
$E_{\min}$	-126	$\leq -1022$	-1022	$\leq -16382$
Exponent bias	127		1023	

**FIGURE 2.2** Format parameter for the IEEE 754 floating point standard.

In the simulation programs of this thesis, only single precision is used.

The last row in Figure 2.2 shows the concept of exponent bias. This indicates the implied range of the exponent of floating number is no longer strictly positive. For example, if single precision with exponent bias of 127 is adapted, a floating point value with exponent bits "10000001<sub>2</sub>", 129<sub>10</sub>, would be  $(129-127)^2 = 2^2$ . Accordingly, if  $e$  is the value of the exponent,  $f$  is the value of the fraction, and  $s$  is the sign of bit, the floating point number is represented as

$$(-1)^s * f * 2^{e-\text{exponent\_bias}} \quad (2.3)$$

The sign bit  $s$  indicates the sign of the floating point number. The positive number has a sign bit of 0, and, the negative number has a sign bit of 1. In a single precision system, the magnitude range is

$$0 < \text{magnitude} < 1.9999999_{10} * 2^{127} \quad (2.4)$$

Several special cases can occur from arithmetic operations. The first case is called "overflow" when the magnitude is greater than the upper limit of the equation (2.4). The second case is when the magnitude is less than  $2^{-126}$ , i.e.

$$0 < \text{magnitude} < 2^{-126}$$

(2.5)

and this is called "underflow". The third situation is how to represent zero, NAN (not a number), and infinity. In the IEEE standard format, the zero is defined as a number with the exponent minimum value and the mantissa zero. The NAN is defined as a number with the exponent being 255. If the single precision is adopted, and the mantissa is not equal to zero, overflow and underflow occurred when the result of an arithmetic operation is beyond or below the representable range [Ref. 6]. However, in the AMD29325 chip only the zero format is the same as that of the IEEE standard. The NAN in the AMD29325 is  $7FA11111_{16}$ , the infinity is  $7FA00000_{16}$ . In this thesis, for reasons of convenience, if all exponent bits are 0, irrespective of the mantissa value, this represents a number  $0_{10}$ . If all bits of a floating point number become 0, it would be the representation of underflow. On the other hand, if all bits except the sign bit are set to 1, it is the representation of infinity.

## B. INTRODUCTION TO FLOATING POINT UNIT CHIP AMD29325

The AMD29325 chip is a high speed floating point processor unit. It performs 32 bits single precision floating point addition, subtraction, multiplication operations in VLSI circuit. It can use the IEEE floating point standard format. The DEC single precision floating point format is also

supported. It includes operations of conversion among 32-bit integer format, floating point format, and IEEE floating point format and DEC floating point format. There are six flags which monitor the status of operations: invalid operation, inexact result, zero, not-a-number(NAN), overflow, and underflow.

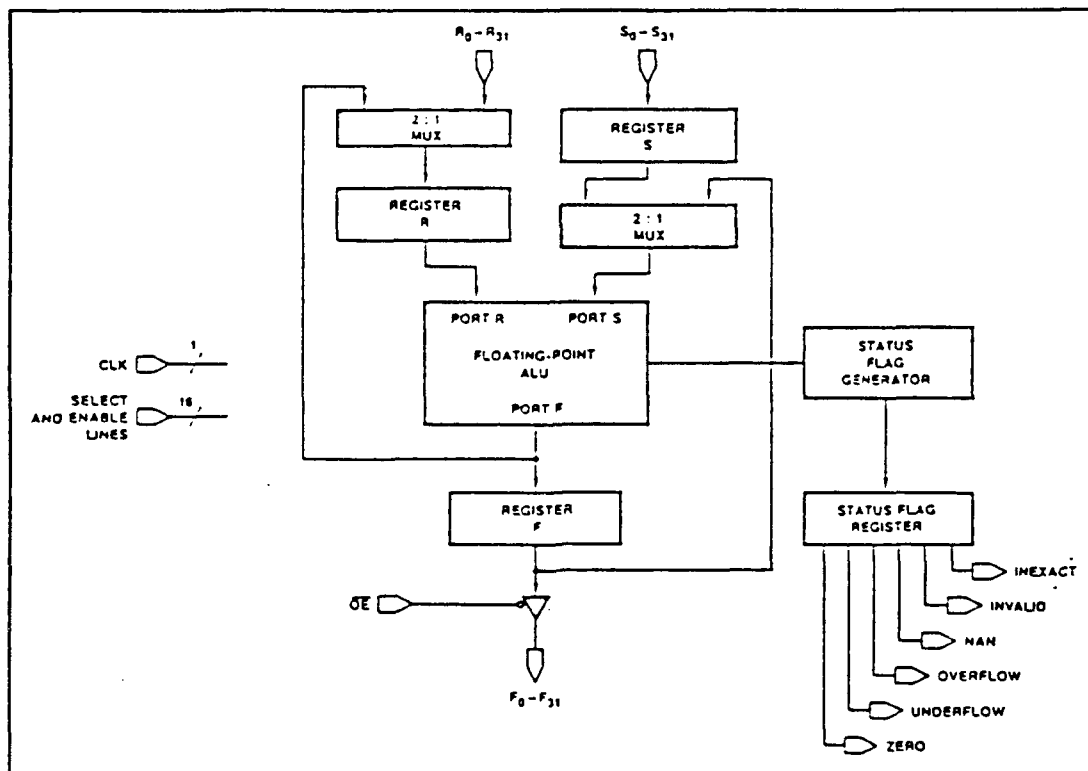
The AMD29325 chip has three buses in 32-bit architecture, two input buses and one output bus. All buses are registered with a clock enable. Input and output registers can be made transparent independently. Figure 2.3 shows the block diagram of the AMD29325. Its pin diagram is shown in Figure 2.4. Selection to perform an arithmetic operation on chip AMD29325 is via the 3 pins  $I_0$ ,  $I_1$ , and  $I_2$ . All selected functions are listed in Figure 2.5.

### **C. BASIC MODELING FUNCTIONS OF AMD29325**

#### **1. THE ELEMENT FUNCTIONS ASSOCIATED WITH THE ARITHMETICAL OPERATION OF AMD29325**

In order to simulate the features of AMD29325, several basic functions had been created before modeling the behavior of the AMD29325. In Figure 2.5, pin  $I_0$ ,  $I_1$ , and  $I_2$  can choose eight different functions. In this thesis, only four arithmetic operations necessary for simulation program had been created; floating point addition, floating point subtraction, floating point multiplication, and floating point division. Although the division function is not used in the





**FIGURE 2.3 AMD29325 block diagram (adopted from AMD data book).**

actual simulation of the AMD29325, it still included in the model of the AMD29325.

The following is a brief description of those element functions associated with the modeling of AMD29325. These element functions are listed in Appendix A.

- **BITSARRAY\_TO\_FP:** to convert the mantissa bits pattern into its corresponding floating point value.
- **FP\_TO\_BITSARRAY:** to do the inverse conversion from floating point value into its corresponding mantissa bits pattern.
- **INT\_TO\_BITSARRAY:** to transfer an integer value into its corresponding bits pattern. Usually, it is used when the exponent value is converted to its corresponding IEEE exponent format.

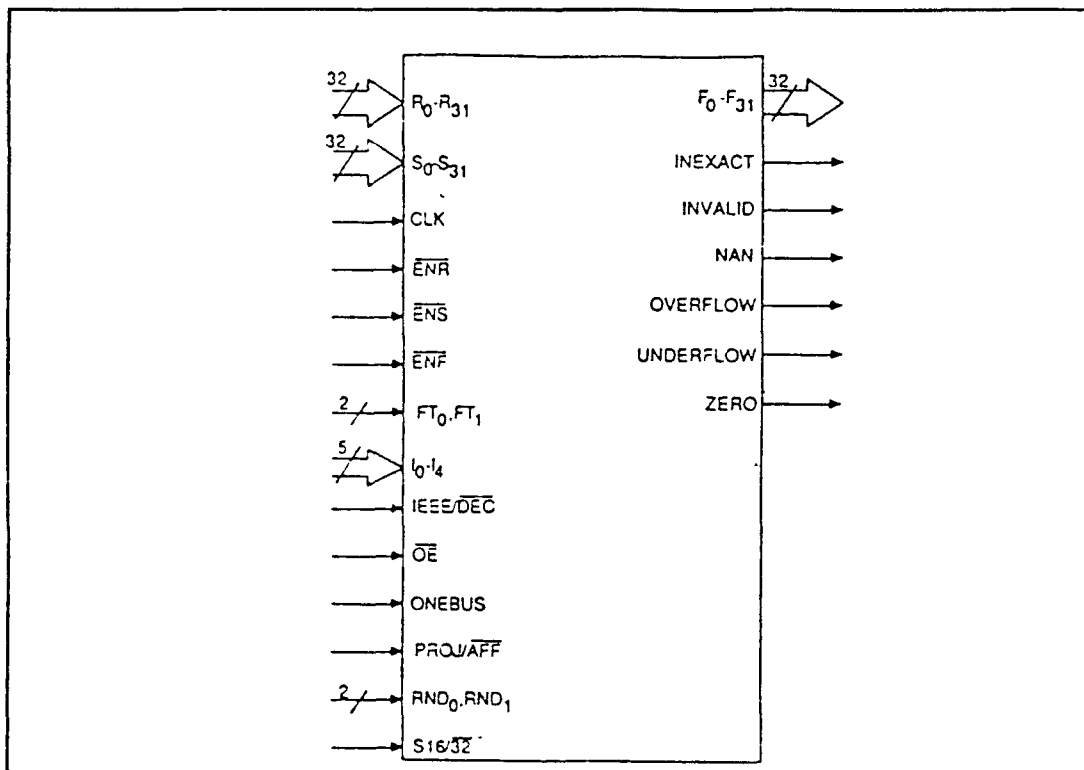


FIGURE 2.4 AMD29325 pin diagram (adopted from the AMD data book).

- UNHIDDEN\_BIT: to recover the hidden bit in the IEEE standard format.
- SHIFL\_TO\_R: to shift the bit pattern from left to right, and the most significant bit is assigned as 0.
- IS\_OVERFLOW: to test the bit pattern of an input parameter to see whether it is overflowed or not.
- IS\_UNDERFLOW: to check the bit pattern of an input parameter to see whether it is underflowed or not.
- IS\_ZERO: to test the bit pattern of an input parameter to see whether it is a zero or not.
- IS\_NAN: to check the bit pattern of an input parameter to see whether it is a NAN expression or not.
- BECOME\_ZERO: to set the result to zero before the actual arithmetic operation occurs. This is a situation of multiplication by zero.

$I_2$	$I_1$	$I_0$	Operation	Output Equation
0	0	0	Floating-point addition (R PLUS S)	$F = R + S$
0	0	1	Floating-point subtraction (R MINUS S)	$F = R - S$
0	1	0	Floating-point multiplication (R TIMES S)	$F = R * S$
0	1	1	Floating-point constant subtraction (2 MINUS S)	$F = 2 - S$
1	0	0	Integer-to-floating-point conversion (INT-TO-FP)	$F \text{ (floating-point)} = R \text{ (integer)}$
1	0	1	Floating-point-to-integer conversion (FP-TO-INT)	$F \text{ (integer)} = R \text{ (floating-point)}$
1	1	0	IEEE-TO-DEC format conversion (IEEE-TO-DEC)	$F \text{ (DEC format)} = R \text{ (IEEE format)}$
1	1	1	DEC-TO-IEEE format conversion (DEC-TO-IEEE)	$F \text{ (IEEE format)} = R \text{ (DEC format)}$

FIGURE 2.5 AMD29325 operation select (adapted from AMD data book).

- **BECOME\_NAN:** to set the result of an operation to be infinity before the actual operation occurs. This is a situation of division by zero.
- **SET\_FLAG:** to verify that the input parameter is located in the representation range which is between the upper limit and lower limit. Otherwise, give it some proper flag if it is not.
- **INCREMENT:** to generate a bit pattern which is greater than input bit pattern by one, For example, output bit pattern is "000111" when the input pattern is "000110".
- **DECREMENT:** to do the inverse as the previous element function.
- **BACK\_TO\_BITARRAY:** to convert a given floating point number into the corresponding IEEE standard bit format.

## 2. THE TOP FUNCTIONS ASSOCIATED WITH THE ARITHMETICAL OPERATIONS OF AMD29325

Four important features of the AMD29325 are created in this thesis. These are the addition function, subtraction function, multiplication function, and division function. The algorithms of these arithmetic functions are described below. These arithmetic functions will call those element functions mentioned previously. All of the VHDL source programs of the arithmetic functions are attached in Appendix B.

### a. Addition Operation Function

Since the operands are in the IEEE standard format, before the addition operation can occur, conversion from IEEE standard bit pattern into a floating point value is necessary. Immediately after the result of this addition operation is generated, conversion of the floating point value back into the IEEE standard format will be done. In the following, the key steps of floating point addition operation are described. Let  $e_i$  and  $s_i$  be used as the exponent and mantissa value of a floating point  $a_i$ . The basic procedure for adding two floating point number  $a_1$  and  $a_2$  is very straight forward and involves four steps.

- (1) if  $e_1$  is less than or equal to  $e_2$ , find the distance  $d$  between  $e_1$  and  $e_2$ . This means  $d$  is equal to  $e_2$  minus  $e_1$ .
- (2) shift  $s_1$  by  $d$  places to the right, now it become  $s_1'$ .
- (3) find the sum of  $s_2$  and  $s_1'$ .

- (4) determine the sign from  $a_2$ , since the absolute value of  $a_2$  is greater than  $a_1$ .

**b. Subtraction Operation Function**

Similar to the addition operation function as mentioned above, the subtraction operation function can be performed by calling the addition operation function after the sign of the minuend has been changed to its inverse.

**c. Multiplication Operation Function**

As mentioned previously, the operands are in the IEEE standard format. Therefore, before this operation function can occur, they are converted into floating point value. Once the result of this multiplication operation is obtained, it is converted back into the IEEE standard format. In the following steps the product of two floating numbers is calculated. Let  $p_1$  and  $e_1$  be the value of mantissa and exponent of  $a_1$  respectively. The method for multiplication of two floating numbers  $a_1$  and  $a_2$  is similar to integer number multiplication.

- (1) find the sum of  $e_1$  and  $e_2$ , and adjust it. If single precision is adopted in the system, the normalized action is the subtraction of 127 from the exponent value.
- (2) find the product of  $p_1$  and  $p_2$ , and adjust it to the range shown in equation (2.2) and modify the adjusted sum of the exponent at the same time.

**d. Division Operation Function**

As mentioned previously, the conversion of the IEEE standard format into floating point format is necessary. When the quotient is generated, it would be converted back into the IEEE standard format. In the following steps the floating

point number division operation is described. Let  $p_i$  and  $e_i$  be the mantissa and exponent of  $a_i$ . Assume that the dividend and divisor are  $a_1$  and  $a_2$  respectively.

(1) find out the distance  $d$  between  $e_1$  and  $e_2$  and then denormalize it. As previous examples, the action of denormalizing means that the distance  $d$  is added to 127.

(2) find out the quotient of the division operation. Then adjust it into the proper range in equation (2.2), and at same time modify the quotient.

### 3. BEHAVIORAL DESCRIPTION OF THE AMD29325 CHIP

As shown in Figure 2.6, an entity of a full adder with port and generic is declared. Generic provides a channel to pass a parameter of constant timing to a component from its environment, and port supports a signal list which is an interface to its environment. 'In' and 'Out' are used to indicate the direction of the signal data flow. In the VHDL language, there are three levels of abstraction possible to

```
entity FULL_ADDER is
    generic( del_1 : TIME := 10 ns ;
             del_2 : TIME := 20 ns ) ;

    port( X, Y, Cin : in BIT ;
          Sum, Cout : out BIT ) ;

end entity FULL_ADDER ;
```

FIGURE 2.6 The entity of a FULL\_ADDER.

### **Behavioral Constructs**

```
architecture behavioral_view of full_adder is
begin
  process
    variable N: integer ;
  constant sum_vector : bit_vector( 0 to 3):="0101";
  constant carry_vector: bit_vector( 0 to 3):="0011";
  begin
    wait on X, Y, Cin ;
    N := 0 ;
    if X = '1' then N := N+1; end if ;
    if Y = '1' then N := N+1; end if ;
    if Cin = '1' then N := N+1 ; end if ;
    Sum <= sum_vector after del_1 ;
    Cout <= carry_vector after del_2 ;
  end process ;
end behavioral_view;
```

### **Data Flow Constructs**

```
architecture dataflow_view of full_adder is
  signal S: bit ;
begin
  S <= X xor Y after del_1 ;
  Sum <= S xor Cin after del_1 ;
  Cout <= (X and Y) or (S and Cin) after del_2;
end dataflow_view;
```

### **Structural Constructs**

```
architecture structure_view of full_adder is
  component half_adder
    generic( delay : time := 0 ns ) ;
    port(l1, l2:in bit;
          C, S: out bit ); end component ;
  component or_gate
    generic( delay : time := 0 ns ) ;
    port(l1, l2:in bit;
          O: out bit ); end component ;
  signal a,b,c :bit ;
begin
  U1: half_adder generic( delay => del_1 );
    port map( X,Y,a,b );
  U2: half_adder generic( delay => del_1 );
    port map( b,Cin,c,Sum );
  U3: or_gate generic( delay => del_2 );
    port map( a,c,Cout );
end structure_view ;
```

**FIGURE 2.7** Three constructs in VHDL language (adopted from [Ref. 4]).

## PIN DESCRIPTION

**R<sub>0</sub>–R<sub>31</sub>** R Operand Bus (Input)  
R<sub>0</sub> is the least-significant bit.

**S<sub>0</sub>–S<sub>31</sub>** S Operand Bus (Input)  
S<sub>0</sub> is the least-significant bit.

**F<sub>0</sub>–F<sub>31</sub>** F Operand Bus (Output)  
F<sub>0</sub> is the least-significant bit.

**CLK** Clock (Input)  
For the internal registers.

**ENR** Register R Clock Enable (Input; Active LOW)  
When  $\overline{\text{ENR}}$  is LOW, register R is clocked on the LOW-to-HIGH transition of CLK. When  $\overline{\text{ENR}}$  is HIGH, register R retains the previous contents.

**ENS** Register S Clock Enable (Input; Active LOW)  
When  $\overline{\text{ENS}}$  is LOW, register S is clocked on the LOW-to-HIGH transition of CLK. When  $\overline{\text{ENS}}$  is HIGH, register S retains the previous contents.

**ENF** Register F Clock Enable (Input; Active LOW)  
When  $\overline{\text{ENF}}$  is LOW, register F is clocked on the LOW-to-HIGH transition of CLK. When  $\overline{\text{ENF}}$  is HIGH, register F retains the previous contents.

**OE** Output Enable (Input; Active LOW)  
When  $\overline{\text{OE}}$  is LOW, the contents of register F are placed on F<sub>0</sub>–F<sub>31</sub>. When  $\overline{\text{OE}}$  is HIGH, F<sub>0</sub>–F<sub>31</sub> assume a high-impedance state.

**ONEBUS** Input Bus Configuration Control (Input)  
A LOW on ONEBUS configures the input bus circuitry for two-input bus operation. A HIGH on ONEBUS configures the input bus circuitry for single-input bus operation.

**FT<sub>0</sub>** Input Register Feedthrough Control (Input; Active HIGH)  
When FT<sub>0</sub> is HIGH, registers R and S are transparent.

**FT<sub>1</sub>** Output Register Feedthrough Control (Input; Active HIGH)  
When FT<sub>1</sub> is HIGH, register F and the status flag register are transparent.

**I<sub>0</sub>–I<sub>2</sub>** Operation Select Lines (Input)  
Used to select the operation to be performed by the ALU. See Table 1 for a list of operations and the corresponding codes.

**I<sub>3</sub>** ALU S Port Input Select (Input)  
A LOW on I<sub>3</sub> selects register S as the input to the ALU S port. A HIGH on I<sub>3</sub> selects register F as the input to the ALU S port.

**I<sub>4</sub>** Register R Input Select (Input)

A LOW on I<sub>4</sub> selects R<sub>0</sub>–R<sub>31</sub> as the input to register R. A HIGH selects the ALU F port as the input to register R.

**IEEE/DEC** IEEE/DEC Mode Select (Input)

When IEEE/DEC is HIGH, IEEE mode is selected. When IEEE/DEC is LOW, DEC mode is selected.

**S16/32** 16- or 32-Bit I/O Mode Select (Input)

A LOW on S16/32 selects the 32-bit I/O mode; a HIGH selects the 16-bit I/O mode. In 32-bit mode, input and output buses are 32 bits wide. In 16-bit mode, input and output buses are 16 bits wide, with the least- and most-significant portions of the 32-bit input and output words being placed on the buses during the HIGH and LOW portions of CLK, respectively.

**RND<sub>0</sub>, RND<sub>1</sub>** Rounding Mode Selects (Input)

RND<sub>0</sub> and RND<sub>1</sub> select one of four rounding modes. See Table 5 for a list of rounding modes and the corresponding control codes.

**PROJ/AFF** Projective/Affine Mode Select (Input)

Choice of projective or affine mode determines the way in which infinities are handled in IEEE mode. A LOW on PROJ/AFF selects affine mode; a HIGH selects projective mode.

**OVERFLOW** Overflow Flag (Output; Active HIGH)

A HIGH indicates that the last operation produced a final result that overflowed the floating-point format.

**UNDERFLOW** Underflow Flag (Output; Active HIGH)

A HIGH indicates that the last operation produced a rounded result that underflowed the floating-point format.

**ZERO** Zero Flag (Output; Active HIGH)

A HIGH indicates that the last operation produced a final result of zero.

**NAN** Not-a-Number Flag (Output; Active HIGH)

A HIGH indicates that the final result produced by the last operation is not to be interpreted as a number. The output in such cases is either an IEEE Not-a-Number (NAN) or a DEC-reserved operand.

**INVALID** Invalid Operation Flag (Output; Active HIGH)

A HIGH indicates that the last operation performed was invalid; e.g.,  $\infty$  times 0.

**INEXACT** Inexact Result Flag (Output; Active HIGH)

A HIGH indicates that the final result of the last operation was not infinitely precise, due to rounding.

FIGURE 2.8 AMD29325 pin description (adopted from the AMD data book).



describe specific circuits [Ref. 7]. In Figure 2.7, examples use three different levels to depict the same full adder as shown. The first way is the behavioral level description, which uses a conditional branch structure in the process. The second way is the data flow level description, which uses the signal assignment statement to express the relationship between input and output. The final way is the structural level description which instantiates several components to build the adder circuit. There are differences among these three levels. Usually, there is a mixed situation where more than one level of abstraction is used in the simulation model. In the program attached in the Appendix, you can find mixed constructs there.

The VHDL simulation program of the chip AMD29325 is attached in Appendix C. In this program, there are four arithmetic functions implemented, floating point addition, floating point subtraction, floating point multiplication, and floating point division. Four flags are checked: not a number(NAN), zero, underflow, and overflow. In order to better understand the usage of the chip pins, the AMD29325 pin description is listed in Figure 2.8. Since many functions of this chip are not required in the simulation for this thesis, those pins are only listed in the port declaration of the AMD29325. A simplified entity A29325 is created, which is attached in the Appendix D. Generally speaking, only those pins of input and output signals, operation functions, clock,

and chip enable necessary for simulation are included in the port declaration of the AMD29325.

When the model is called by the other top level environment, the two input signal buses must be driven and the chip enable signal must be active low. When the clock comes with the positive rising edge, the floating point unit is triggered to execute the selected operation function. Data on the output bus will change after a constant time delay. Since the constant time delay is the VHDL inertial delay, the desired output data will be preempted and not shown on the data bus, this is the situation when the period of the clock is less than the constant delay of the selected operation. When the floating point unit AMD29325 is employed in a system design, it is necessary to be sure that the period of the clock is greater than the constant delay of the chip. Otherwise, undesired output data signals may appear on the output data bus.

All element functions, arithmetic functions, and the total behavior of the AMD29325 have been introduced in this chapter. In the next chapter, the subject will focus on the system configuration.

### III. THE DATA FLOW DESIGN OF THE FAST FOURIER TRANSFORM

#### A. OVERVIEW OF THE FAST FOURIER TRANSFORM

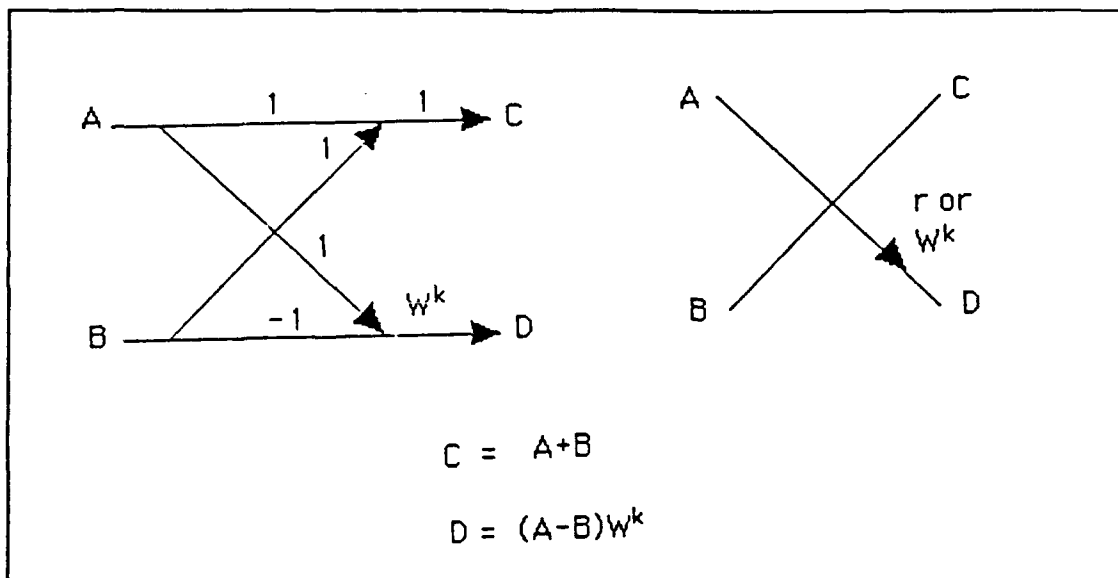
The Fourier Transform is usually used to change time domain data into frequency domain data for spectral analysis. For some problems the analysis in the frequency domain is simpler than that in the time domain. For Discrete Fourier Transform(DFT), the operations are performed on a sequence of data. Assume that the total number of input data is  $N$ , which is an integer of power of 2. For a limited sequence  $x(n)$ , the Discrete Fourier transform formula is,

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N} \quad \text{for } k=0 \dots N-1 \quad (3.1)$$

In the following a brief description of two data flow designs of Fast Fourier Transform are presented. They are the methods of decimation in time and decimation in frequency.

##### 1. DECIMATION IN TIME(DIT)

In this method, it is possible to divide  $x(n)$  into two half series. One with odd sequence number, and the other with even sequence number. Through a well known derivation of steps, the butterfly operation for the DIT fast fourier transform can be represented graphically in Figure 3.1 [Ref. 8]. The complete signal flow of an 8-point FFT is shown in Figure 3.2 [Ref. 1]. Note that in this figure the input

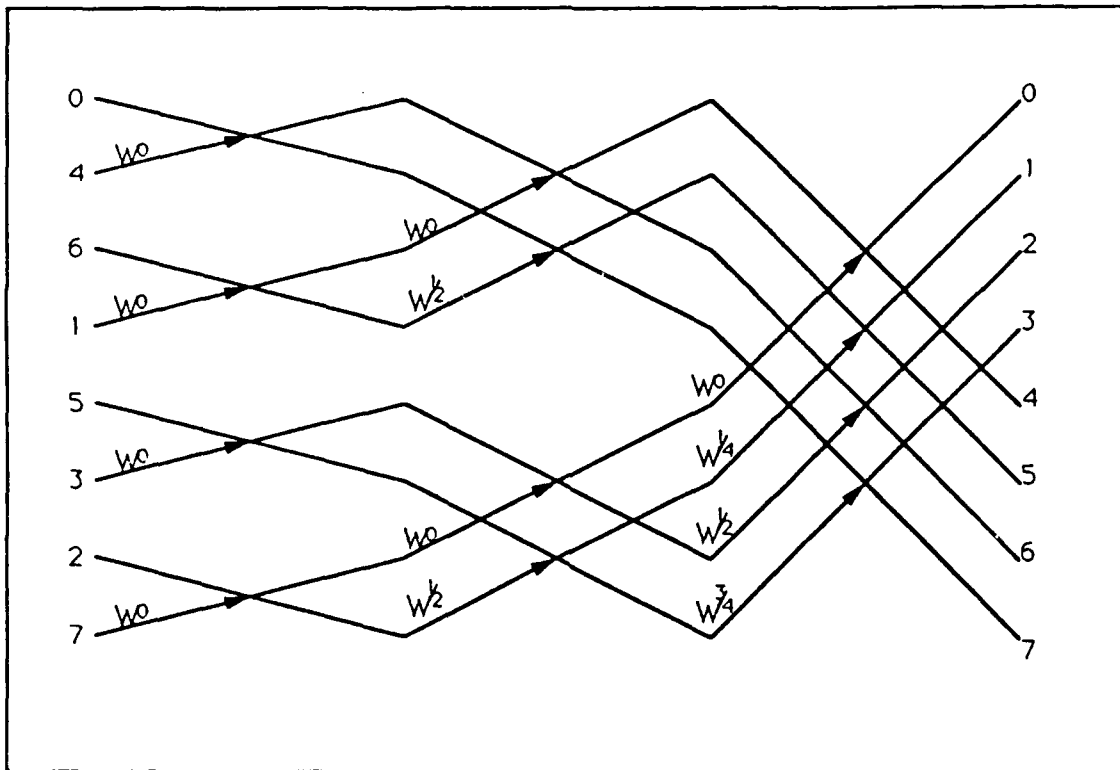


**FIGURE 3.1** Signal flow graph and the shorthand representation of DIT butterfly.

data is arranged in bit reversal order according to the needs of decimation. This arrangement has the property that the output will turn out to be in the natural order.

## 2. DECIMATION IN FREQUENCY(DIF)

Another way to decompose the calculation of the Discrete Fourier Transform(DFT) is known as the decimation in frequency. This idea is similar to the idea of the decimation in time. In DIT, the time sequence was partitioned into two subsequences having even and odd indices. An alternative is to partition the time sequence  $x(n)$  into first and second halves. The signal data flow of the butterfly is shown in Figure 3.3 [Ref. 1]. And the completed signal data flow of an 8-point FFT in DIF algorithm is shown in Figure 3.4 [Ref. 1]. Figure 3.4 is similar to Figure 3.2, except that bit reversal ordering



**FIGURE 3.2 8 points FFT using DIT butterfly.**

occurred in the output. In both Figure 3.2 and Figure 3.4, two data values are used as a pair inputs to a butterfly calculation. The output can be put back into the same storage locations that hold the initial input values because they are no longer needed for any subsequent computations. As a consequence of this characteristic, the FFT shown in Figure 3.2 and 3.3 are called in-place algorithm. Another arrangement is to have both the input and the output data in the normal order. Figure 3.5 shows a non-bit-reversal algorithm. Notice that this is no longer an in-place algorithm. In this thesis, in order to keep normal order for both the input and the output data, the non-in-place algorithm is adopted.

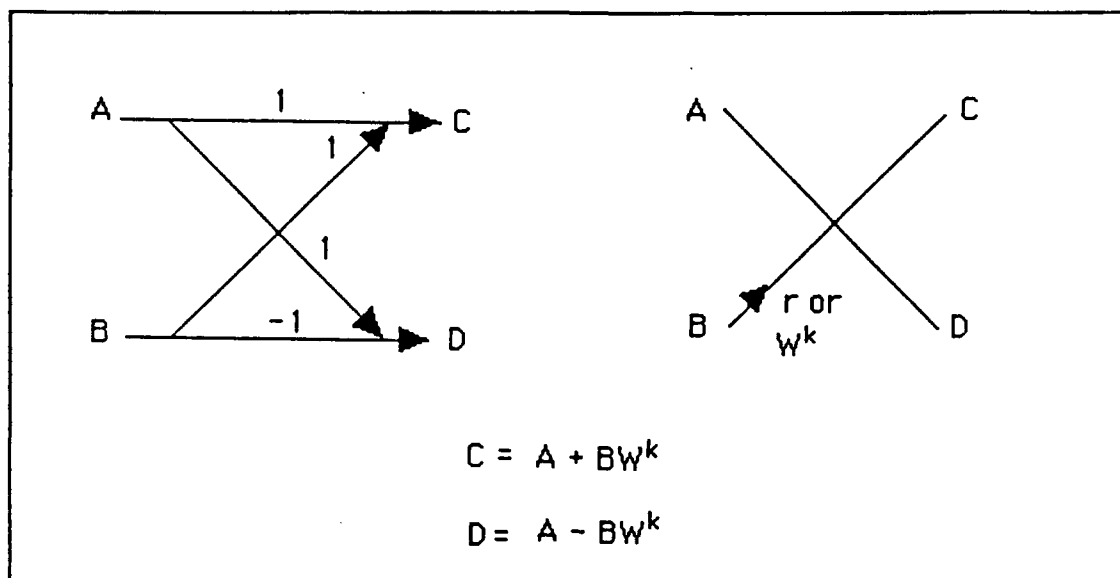


FIGURE 3.3 Signal flow graph and shorthand representation in DIF butterfly.

#### B. COMPARISON OF SEVERAL DATA FLOW CONFIGURATIONS OF THE FAST FOURIER TRANSFORM

The objective is to consider several data flow structures to find an optimum implementation of the Fast Fourier Transform. Figure 3.6 shows the basic butterfly structures of both the DIT and the DIF Fast Fourier Transform. There are two inputs, complex numbers A and B. They are combined together with a complex weight factor,  $w^k$ , to form two outputs C and D. Inspection of the formula shows that a single butterfly calculation requires one complex addition, one complex subtraction, and one complex multiplication. Additionally, five complex memory access are required; three reads for A, B and  $w^k$ , and two writes for C and D. Figure 3.6 shows the total

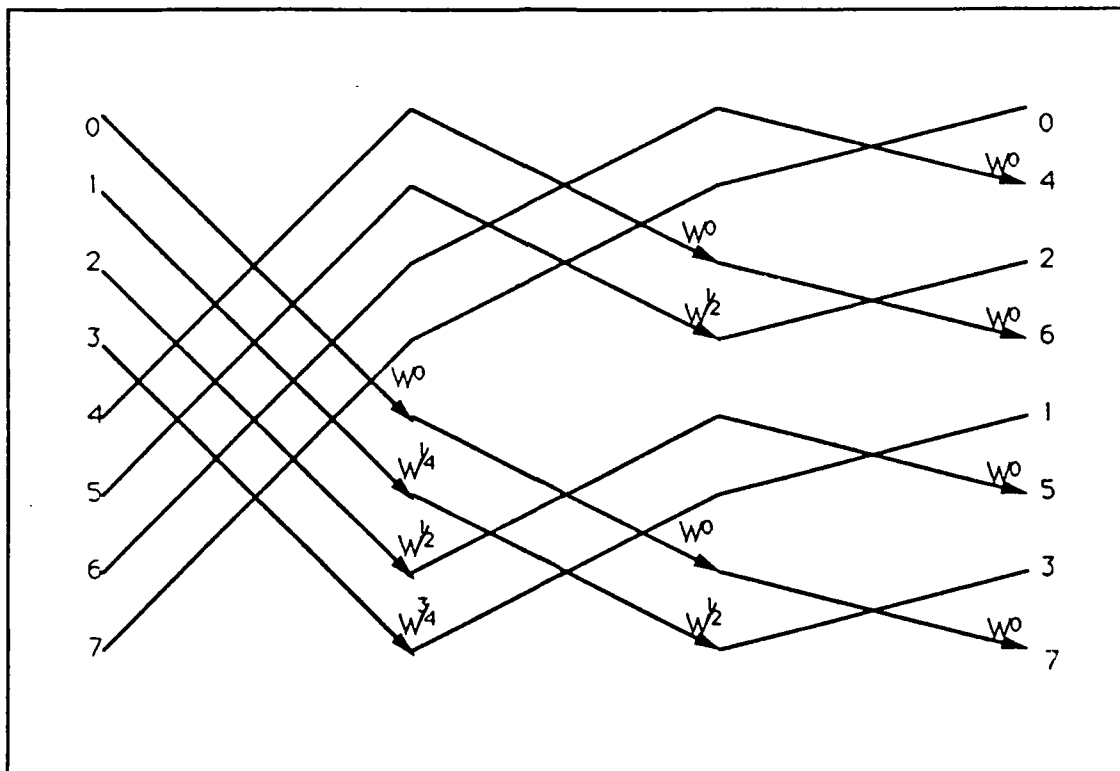


FIGURE 3.4 8 points FFT using DIF butterfly.

number of floating point operations, data read, data write, and coefficient read required.

From the above analysis it is known that if all operations take equal time, the throughput is limited by the memory access requirement. In order to ease this bottleneck, two ways were adopted. Firstly, the real and the imaginary parts of the input complex data are accessed simultaneously. Secondly, it is noted that the multiplications are performed between the data and a coefficient. If the coefficients are stored in a separate memory, they may be accessed concurrently. Several different structures associated with a non-in-place algorithm of the butterfly in the DIF are discussed below.

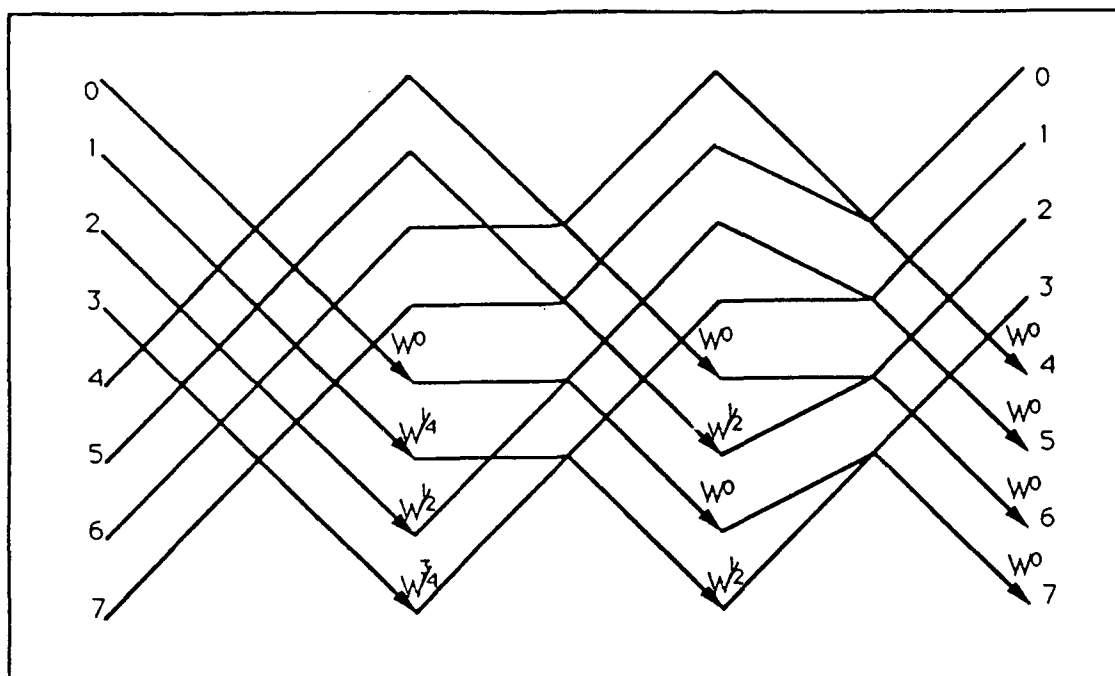
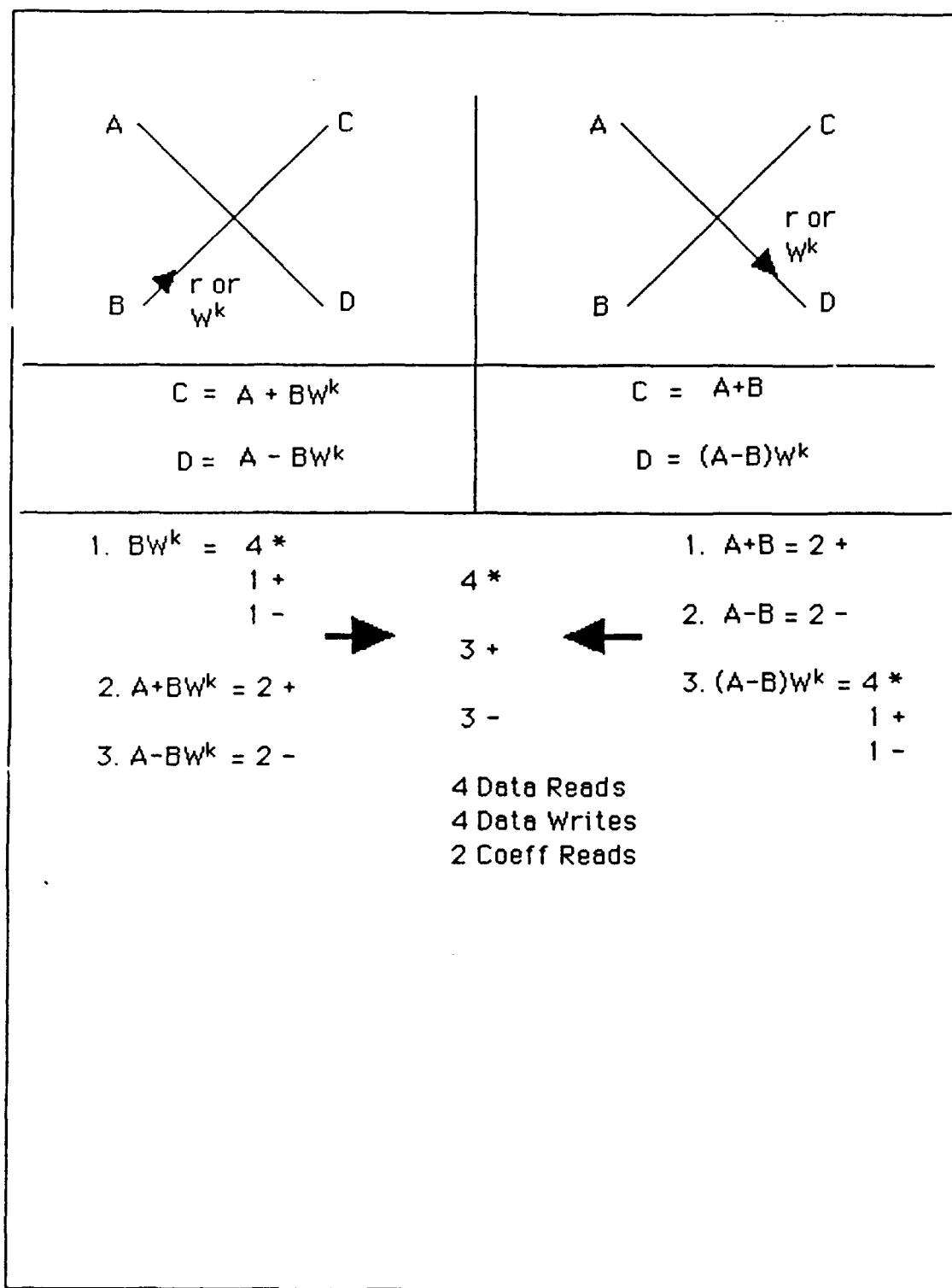


FIGURE 3.5 8 points FFT with DIF butterfly in non-bit-reversal algorithm.

### 1. STRUCTURE 1 OF DIF BUTTERFLY

It is known that the total number of required arithmetic operations for real data is 10, which includes four multiplications and six additions/subtractions. In order to reduce the execution steps, a full pipeline structure can be adopted. In this full pipeline structure shown in Figure 3.7, each arithmetic operation uses a processor. Therefore, for a total number of 10 arithmetic operations, it needs 10 processors. The data flow configuration is shown in Figure 3.7. There are three layers of arithmetic processors shown. There is one layer for data read, and one layer for data write not shown in Figure 3.7. The time space diagram for this





**FIGURE 3.6 Two different basic butterflies and their arithmetic operations.**

structure is listed in Table 3.1. For data sample  $A(n)$ ,  $B(n)$ , and  $W^k(n)$  the complete butterfly operation needs 5 time steps. These steps are shown in shaded boxes in Table 3.1. At the  $N$ th time step the input data  $A$ ,  $B$ , and  $W^k$  are fetched. In the next 3 time steps, the output data  $C$  and  $D$  are generated. At the time step  $N+4$ ,  $C$  and  $D$  are stored back to memory. Four steps of data flow execution can be overlapped with the execution of the previous data. Since in this thesis single precision IEEE floating point format (32 bits) is used, the total size of the input data and output data buses are 192 and 128 respectively which are shown in Figure 3.7. This structure requires input and output buses concurrently. Therefore, time multiplexed buses by input and output are not usable in this structure. Because input and output buses are always busy, the bus utility of this structure is 100% as shown in Table 3.1. Every processor in this structure is always busy, therefore, the average efficiency of processors is 100%. The average efficiency of processors is defined as the percentage of processors used in one completed cycle of the arithmetic time space table. For example, in structure 1, since all 10 processes are busy in one row of the time space table, the

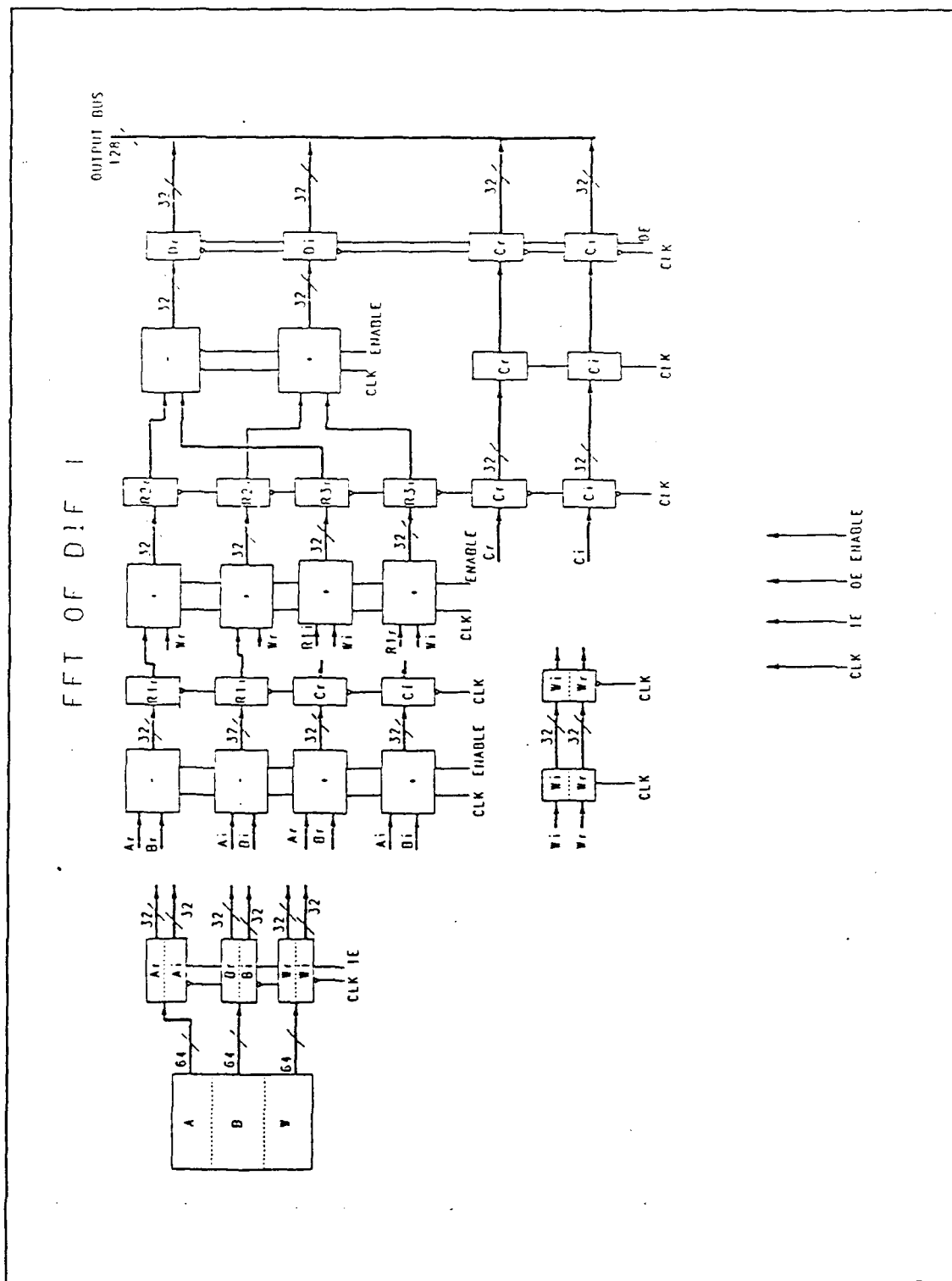


FIGURE 3.7 Butterfly implementation in pipeline structure.

S t e p	O u t p u t  B u s	I n p u t  B u s	1st row ALU's oper. for "+" & "-"	2nd row ALU's oper. for "*" & "/"	3rd row ALU's oper. for "+" & "-"
N	C(n-4) D(n-4)	B(n) W(n) A(n)	$Cr(n-1) = Ar(n-1) + Br(n-1)$ $R1r(n-1) = Ar(n-1) - Br(n-1)$ $Ci(n-1) = Ai(n-1) + Bi(n-1)$ $R1i(n-1) = Ai(n-1) - Bi(n-1)$	$R2r(n-2) = R1r(n-2) * Wr(n-2)$ $R2i(n-2) = R1r(n-2) * Wi(n-2)$ $R3r(n-2) = R1i(n-2) * Wi(n-2)$ $R3i(n-2) = R1i(n-2) * Wr(n-2)$	$Dr(n-3) = R2r(n-3) - R3r(n-3)$ $Di(n-3) = R2i(n-3) + R3i(n-3)$
N+1	C(n-3) D(n-3)	B(n+1) W(n+1) A(n+1)	$Cr(n) = Ar(n) + Br(n)$ $R1r(n) = Ar(n) - Br(n)$ $Ci(n) = Ai(n) + Bi(n)$ $R1i(n) = Ai(n) - Bi(n)$	$R2r(n-1) = R1r(n-1) * Wr(n-1)$ $R2i(n-1) = R1r(n-1) * Wi(n-1)$ $R3r(n-1) = R1i(n-1) * Wi(n-1)$ $R3i(n-1) = R1i(n-1) * Wr(n-1)$	$Dr(n-2) = R2r(n-2) - R3r(n-2)$ $Di(n-2) = R2i(n-2) + R3i(n-2)$
N+2	C(n-2) D(n-2)	B(n+2) W(n+2) A(n+2)	$Cr(n+1) = Ar(n+1) + Br(n+1)$ $R1r(n+1) = Ar(n+1) - Br(n+1)$ $Ci(n+1) = Ai(n+1) + Bi(n+1)$ $R1i(n+1) = Ai(n+1) - Bi(n+1)$	$R2r(n) = R1r(n) * Wr(n)$ $R2i(n) = R1r(n) * Wi(n)$ $R3r(n) = R1i(n) * Wi(n)$ $R3i(n) = R1i(n) * Wr(n)$	$Dr(n-1) = R2r(n-1) - R3r(n-1)$ $Di(n-1) = R2i(n-1) + R3i(n-1)$

TABLE 3.1 Time space diagram of DIF structure 1.

N + 3	C(n-1)	B(n+3)	Cr(n+2) =	R2r(n+1) =	Dr(n) =
	D(n-1)	W(n+3)	Ar(n+2)+Br(n+2)	R1r(n+1)*Wr(n+1)	R2r(n) -
		A(n+3)	R1r(n+2) =	R2i(n+1) =	R3r(n)
			Ar(n+2)-Br(n+2)	R1r(n+1)*Wi(n+1)	Di(n) =
			Ci(n+2) =	R3r(n+1) =	R2i(n) +
			Ai(n+2)+Bi(n+2)	R1i(n+1)*Wi(n+1)	R3i(n)
			R1i(n+2) =	R3i(n+1) =	
			Ai(n+2)-Bi(n+2)	R1i(n+1)*Wr(n+1)	
N + 4	C(n)	B(n+4)	Cr(n+3) =	R2r(n+2) =	Dr(n+1) =
	D(n)	W(n+4)	Ar(n+3)+Br(n+3)	R1r(n+2)*Wr(n+2)	R2r(n+1) -
		A(n+4)	R1r(n+3) =	R2i(n+2) =	R3r(n+1)
			Ar(n+3)-Br(n+3)	R1r(n+2)*Wi(n+2)	Di(n+1) =
			Ci(n+3) =	R3r(n+2) =	R2i(n+1) +
			Ai(n+3)+Bi(n+3)	R1i(n+2)*Wi(n+2)	R3i(n+1)
			R1i(n+3) =	R3i(n+2) =	
			Ai(n+3)-Bi(n+3)	R1i(n+2)*Wr(n+2)	
N + 5	C(n+1)	B(n+5)	Cr(n+4) =	R2r(n+3) =	Dr(n+2)=
	D(n+1)	W(n+5)	Ar(n+4)+Br(n+4)	R1r(n+3)*Wr(n+3)	R2r(n+2) -
		A(n+5)	R1r(n+4) =	R2i(n+3) =	R3r(n+2)
			Ar(n+4)-Br(n+4)	R1r(n+3)*Wi(n+3)	Di(n+2) =
			Ci(n+4) =	R3r(n+3) =	R2i(n+2) +
			Ai(n+4)+Bi(n+4)	R1i(n+3)*Wi(n+3)	R3i(n+2)
			R1i(n+4) =	R3i(n+3) =	
			Ai(n+4)-Bi(n+4)	R1i(n+3)*Wr(n+3)	

input bus size = 192 bits; output bus size = 128 bits

# of execution steps per data sample = 5

# of overlapped steps in two adjacent data samples = 4

average efficiency of processors = 100 %

bus utility = 100 %

TABLE 3.1 Time space diagram of DIF structure 1(continued).

average efficiency of the processors in this structure is 100%.

## 2. STRUCTURE 2 OF DIF BUTTERFLY

For structure 1, the disadvantage is that the number of input and output data buses is too large. Here, in structure 2 the number of I/O data lines required is reduced. 6 processors are used to implement a butterfly structure in Figure 3.8, 2 for subtraction or addition and 4 for multiplication. Due to the time multiplexing, the sizes of the input and output buses are decreased to 128. An overlap time space diagram is listed in Table 3.2. In Figure 3.8,  $R2i$ ,  $R3i$ ,  $R2r$  and  $R3r$  are fed back to the first row processors through the selectors controlled by the selection signal  $S1$ . Therefore, the data flow sequence controller of this structure will be more complicated than that of structure 1. In Figure 3.8, extra registers are used to stored the previous input data  $A(n)$ . When the current data  $A(n+1)$  is read, the processors need to get the previous input data  $A(n)$ ,  $B(n)$  and  $W(n)$  for the arithmetic operations concurrently. Therefore, a second pair of registers is used here as a buffer to save the previous input data  $A$ . The number of time steps for a data sample is 6, while in structure 1 only 5 were required. The number of overlap time steps for two adjacent data samples is 3. In Table 3.2, the number of rows for one cycle of arithmetic operation in the time space is 3, which means that

all of the arithmetic operations will be repeated at every 3 time steps. From step  $N$  to  $N+2$ , there are 6 times space boxes and only 4 boxes are used by processors. The multiplication is performed in 1 of every 3 steps. The operations for the multiplier in the box is 4. The total number of operations in those 6 boxes should be 18, but only 10 operations are executed. Therefore, the average efficiency of processors is 56%.

Although the number of data bus lines is reduced, the data bus utility, which is 83%, is decreased by 17% compared with that of structure 1. This results from the fact that from step  $N$  to  $N+2$  the time space boxes associated with data buses are 6, and only 5 boxes were used to convey data. Here, it is not allowed to use time multiplexed buses for both input and output, because the input bus is always busy.

### **3. STRUCTURE 3 OF DIF BUTTERFLY**

In structure 2, the average efficiency of processors was 56% which is lower than that of structure 1. In structure 3, the emphasis is to increase the processor operation efficiency. There are four processors arranged to perform different arithmetic operations at different times in structure 3. The performance of this structure is better than that of the structure 2. In Figure 3.9, more selectors than that of structure 2 are used. The input data is fed at the proper time to the floating point unit(FPU) by selection





signals S1 and S2. However, the method for generating the

S t e p	Output Bus	Input Bus	1st row ALU's oper. for "+" & "-"	2nd row Multipliers
N		A(n)	$Cr(n-1) = Ar(n-1) + Br(n-1)$  $Ci(n-1) = Ai(n-1) + Bi(n-1)$	$R2r(n-1) = R1r(n-1) * Wr(n-1)$  $R2i(n-1) = R1r(n-1) * Wi(n-1)$  $R3r(n-1) = R1i(n-1) * Wi(n-1)$  $R3i = R1i(n-1) * Wr(n-1)$
N + 1	C(n-1)	B(n)	$Dr(n-1) = R2r(n-1) - R3r(n-1)$  $Di(n-1) = R2i(n-1) + R3i(n-1)$	
N + 2	D(n-1)	W(n)	$R1r(n) = Ar(n) - Br(n)$  $R1i(n) = Ai(n) - Bi(n)$	
N + 3		A(n+1)	$Cr(n) = Ar(n) + Br(n)$  $Ci(n) = Ai(n) + Bi(n)$	$R2r(n) = R1r(n) * Wr(n)$  $R2i(n) = R1r(n) * Wi(n)$  $R3r(n) = R1i(n) * Wi(n)$  $R3i(n) = R1i(n) * Wr(n)$
N + 4	C(n)	B(n+1)	$Dr(n) = R2r(n) - R3r(n)$  $Di(n) = R2i(n) + R3i(n)$	

TABLE 3.2 Time space diagram of DIF structure 2.

N + 5	D(n)	W(n+1)	$R1r(n+1) =$ $Ar(n+1) - Br(n+1)$  $R1i(n+1) =$ $Ai(n+1) - Bi(n+1)$	
N + 6		A(n+2)	$Cr(n+1) =$ $Ar(n+1) + Br(n+1)$  $Ci(n+1) =$ $Ai(n+1) + Bi(n+1)$	$R2r(n+1) =$ $R1r(n+1) * Wr(n+1)$  $R2i(n+1) =$ $R1r(n+1) * Wi(n+1)$  $R3r(n+1) =$ $R1i(n+1) * Wi(n+1)$  $R3i(n+1) =$ $R1i(n+1) * Wr(n+1)$

input bus size = 64 bits

output bus size = 64 bits

# of execution steps per data sample = 6

# of overlap steps for two adjacent data samples = 3

average efficiency of processors = 56 %

bus utility = 83 %

**TABLE 3.2 Time space diagram of DIF structure 2(continued).**

selection signals and which functional signals F1 through F5 should be generated in this structure are important issues. In Table 3.3, the input data samples  $A(n)$ ,  $B(n)$ , and  $W(n)$  to be manipulated are shadowed in this table. The functional signals F1 through F5 are used to change the processors to the correct arithmetic function at the right time.

The processor average efficiency of this structure is higher than that of structure 2. It still has the same 2 empty time space boxes as structure 2 in row  $N$  to  $N+2$  as shown in Table 3.3. However, the number of operations associated with the boxes in this structure is 1. The complete cycle of butterfly operations is 3. The number of arithmetic operations in 3 rows should be 12, but the number of actual operations is 10. Therefore, the average of efficiency of the processors is 83%. It is higher than that of structure 2, but is still lower than that of structure 1. As a matter of fact, the size of data bus lines, execution steps, and bus utility are the same as those of structure 2. From Table 3.3 and 3.2, it is clear that the environmental support to processors in structure 3 is about the same as that of structure 2, except that a different number of processors are used. Hence, although fewer processors are used than the previous structure, it always keep these processors busy.



Step	Output Bus	Input Bus	Processor # 1	Processor # 2	Processor #3	Processor #4
N	C(n-1)	A(n)	$R2r(n-1) = R1r(n-1) * Wr(n-1)$	$R2i(n-1) = R1r(n-1) * Wi(n-1)$	$R1r(n-1) = R1i(n-1) * Wi(n-1)$	$R1i(n-1) = R1i(n-1) * Wr(n-1)$
N+1		B(n)	$Dr(n-1) = R2r(n-1) - R1r(n-1)$	$Di(n-1) = R1i(n-1) + R2i(n-1)$		
N+2	D(n-1)	W(n)	$Cr(n) = Ar(n) + Br(n)$	$Ci(n) = Ai(n) + Bi(n)$	$R1r(n) = Ar(n) - Br(n)$	$R1i(n) = Ai(n) - Bi(n)$
N+3	C(n)	A(n+1)	$R2r(n) = R1r(n) * Wr(n)$	$R2i(n) = R1r(n) * Wi(n)$	$R1r(n) = R1i(n) * Wi(n)$	$R1i(n) = R1i(n) * Wr(n)$
N+4		B(n+1)	$Dr(n) = R2r(n) - R1r(n)$	$Di(n) = R1i(n) + R2i(n)$		
N+5	D(n)	W(n+1)	$Cr(n+1) = Ar(n+1) + Br(n+1)$	$Ci(n+1) = Ai(n+1) + Bi(n+1)$	$R1r(n+1) = Ar(n+1) - Br(n+1)$	$R1i(n+1) = Ai(n+1) - Bi(n+1)$
N+6	C(n+1)	A(n+2)	$R2r(n+1) = R1r(n+1) * Wr(n+1)$	$R2i(n+1) = R1r(n+1) * Wi(n+1)$	$R1r(n+1) = R1i(n+1) * Wi(n+1)$	$R1i(n+1) = R1i(n+1) * Wr(n+1)$

input bus size = 64 bits; output bus size = 64 bits

# of execution steps per data sample = 6

# of overlapped steps in two adjacent data samples = 3

average efficiency of processors = 83%; bus utility = 83%

**TABLE 3.3 Time space diagram of DIF structure 3.**

#### 4. STRUCTURE 4 OF DIF BUTTERFLY

In the previous structure, not every processor is busy all the times. If it is desired to keep the processors busy as in structure 1, and to use fewer processors than in of structure 1, what can be done? In structure 4. Only two processors are used as shown in Figure 3.10. A special device "1:4 DMUX" are used to route the output of the ALU to different buffers. The time space diagram is shown in Table 3.4. In Table 3.4, two processors are always busy. In other words, the average efficiency of processors is 100%, the same as that of structure 1. 8 steps are needed for completing one butterfly operation, and the number of overlapped steps is 3 for two adjacent data samples. The sizes of the input and output data buses are still 64. It is noted that in this structure the bus time space usage repeats every 5 time steps. There is only about 50% usages from step  $N+3$  to step  $N+7$ . This situation can be improved using the time multiplexed bus for input and output to achieve a higher bus utility. In this situation, the controller and address sequence generator for this structure would be more complicated than that of the former structures.



Step	Output Bus	Input Bus	Processor #1	Processor #2
N		A(n)	$R3r(n-1) = R1i(n-1) * Wi(n-1)$	$R3i(n-1) = R1i(n-1) * Wr(n-1)$
N+1		B(n)	$Dr(n-1) = R2r(n-1) - R3r(n-1)$	$Di(n-1) = R2i(n-1) + R3i(n-1)$
N+2	D(n-1)		$Cr(n) = Ar(n) + Br(n)$	$Ci(n) = Ai(n) + Bi(n)$
N+3	C(n)	W(n)	$R1r(n) = Ar(n) - Br(n)$	$R1i(n) = Ai(n) - Bi(n)$
N+4			$R2r(n) = R1r(n) * Wr(n)$	$R2i(n) = R1i(n) * Wi(n)$
N+5		A(n+1)	$R3r(n) = R1i(n) * Wi(n)$	$R3i(n) = R1i(n) * Wr(n)$
N+6		B(n+1)	$Dr(n) = R2r(n) - R3r(n)$	$Di(n) = R2i(n) + R3i(n)$
N+7	D(n)		$Cr(n+1) = Ar(n+1) + Br(n+1)$	$Ci(n+1) = Ai(n+1) + Bi(n+1)$
N+8	C(n+1)	W(n+1)	$R1r(n+1) = Ar(n+1) - Br(n+1)$	$R1i(n+1) = Ai(n+1) - Bi(n+1)$

input bus size = 64 bits

output bus size = 64 bits

# of execution steps per data sample = 8

# of overlapped steps in two adjacent data samples = 3

average efficiency of processors = 100 %

bus utility = 100 %

**Table 3.4 Time space diagram of DIF structure 4.**



## 5. STRUCTURE 5 OF DIF BUTTERFLY

If only a single processor is allowed in the butterfly structure, what would happen? In the following, the emphasis is on using a single processor in the butterfly structure. In DIF Figure 3.5 the total number of arithmetic operation is 10, 4 for multiplication, 6 for additions or subtractions. Additionally, the input data must be fetched and the output data must be stored. An alternative configuration is shown in Figure 3.11 where input data is selected for the FPU, and the output data from FPU is stored to registers selected by the control signals S1 through S6. The selection signals depend on activities shadowed in Table 3.5. In Table 3.5, the total number of steps needed for one butterfly cycle is 13. From step N to step N+12, it still needs a data size of 64 in both input and output buses. However, it is true that the bus utility of 25% is lower than any of the previous structures. The bus activities cycles every 10 steps. From step N+4 to N+13, the total number of time step boxes is 20, but only 5 boxes are used. In order to increase the bus utility, it is necessary to use a time multiplexed bus. One of the disadvantages in this structure is that the real part and the imaginary part of the data can not be manipulated in a single processor simultaneously. Therefore, the imaginary part of the input data must wait until the real part manipulation has been completed.

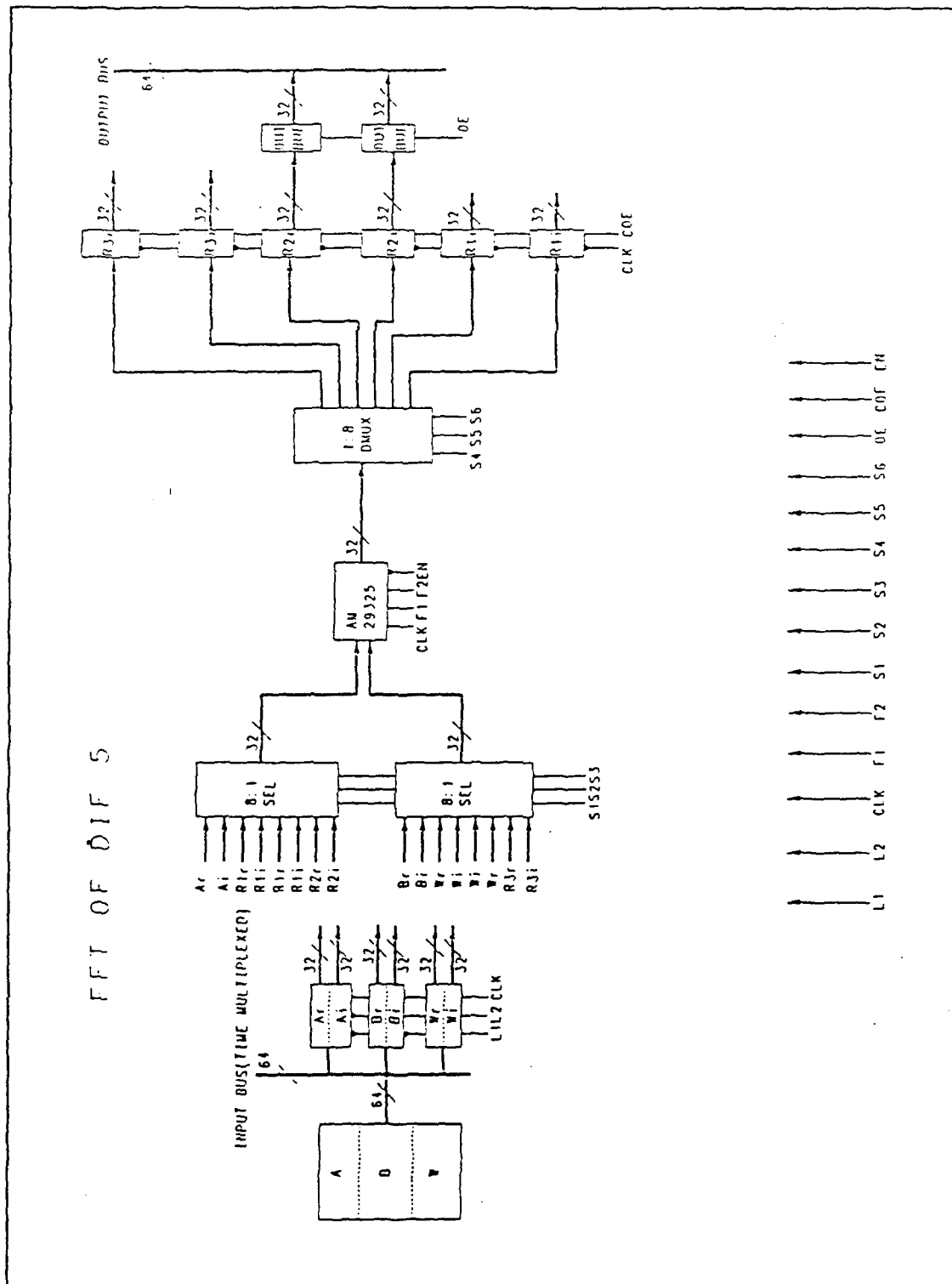


FIGURE 3.11 Butterfly implementation in structure 5.

Step	output bus	input bus	processor
N		A(n)	$Dr(n-1) = R2r(n-1) - R3r(n-1)$
N+1		B(n)	$Di(n-1) = R2i(n-1) + R3i(n-1)$
N+2			$Cr(n) = Ar(n) + Br(n)$
N+3			$Ci(n) = Ai(n) + Bi(n)$
N+4	C(n)		$R1r(n) = Ar(n) - Br(n)$
N+5		W(n)	$R1i(n) = Ai(n) - Bi(n)$
N+6			$R2r(n) = R1r(n) * Wr(n)$
N+7			$R3r(n) = R1i(n) * Wi(n)$
N+8			$R2i(n) = R1r(n) * Wi(n)$
N+9			$R3i(n) = R1i(n) * Wr(n)$
N+10		A(n+1)	$Dr(n) = R2r(n) - R3r(n)$
N+11		B(n+1)	$Di(n) = R2i(n) + R3i(n)$
N+12	D(n)		$Cr(n+1) = Ar(n+1) + Br(n+1)$
N+13			$Ci(n+1) = Ai(n+1) + Bi(n+1)$
N+14	C(n+1)		$R1r(n+1) = Ar(n+1) - Br(n+1)$
N+15		W(n+1)	$R1i(n+1) = Ai(n+1) - Bi(n+1)$
N+16			$R2r(n+1) = R1r(n+1) * Wr(n+1)$
N+17			$R3r(n+1) = R1i(n+1) * Wi(n+1)$
N+18			$R2i(n+1) = R1r(n+1) * Wi(n+1)$
N+19			$R3i(n+1) = R1i(n+1) * Wr(n+1)$
N+20		A(n+2)	$Dr(n+1) = R2r(n+1) - R3r(n+1)$
N+21		B(n+2)	$Di(n+1) = R2i(n+1) + R3i(n+1)$

input bus size = 64 bits;

output bus size = 64 bits

# of execution steps per data sample = 13

# of overlapped steps in two adjacent data samples = 3

average efficiency of processors = 100%; bus utility = 25%

**TABLE 3.5 Time space diagram of DIF structure 5.**

## 6. STRUCTURE 6 OF DIF BUTTERFLY

This structure is a modified version of structure 5 shown in Figure 3.12. The time space diagram is shown in Table 3.6. The bus utility calculation is similar to the previous approach, with only 9 boxes used for every 20 boxes of input/output data. The bus utility is 45% in this structure, which is higher than the previous one. In Table 3.6, it is obvious that the size of the input and output buses are decreased to 32 respectively. The bus utility of this structure is still much lower than that of the structure 1, which was 100%. The bus utility of structure 2 and 3 were 83%. Increase of the buses utility by time multiplexing is achieved at the expense of more complicated controller and address sequence generator. The controller must know whether the current data on bus is input data or output data.

All 6 structures have been introduced, and the comparison is listed in Table 3.7. In this thesis, only the address sequence generator and controller of structure 1 are implemented. In the following section, a design of a controller and addressing sequencer of structure 1 will be presented.



Step	Output Bus	Input Bus	processor
N		$Ar(n)$	$Dr(n-1) = R2r(n-1) - R3r(n-1)$
N+1		$Br(n)$	$Di(n-1) = R2i(n-1) + R3i(n-1)$
N+2		$Ai(n)$	$Cr(n-1) = Ar(n-1) + Br(n-1)$
N+3	$Cr(n)$	$Bi(n)$	$R1r(n-1) = Ar(n-1) - Br(n-1)$
N+4			$Ci(n) = Ai(n) + Bi(n)$
N+5	$Ci(n)$	$Wr(n)$	$R1i(n) = Ai(n) - Bi(n)$
N+6		$Wi(n)$	$R2r(n) = R1r(n) * Wr(n)$
N+7			$R3r(n) = R1i(n) * Wi(n)$
N+8			$R2i(n) = R1i(n) * Wr(n)$
N+9			$R3i(n) = R1r(n) * Wi(n)$
N+10		$Ar(n+1)$	$Dr(n) = R2r(n) - R3r(n)$
N+11	$Dr(n)$	$Br(n+1)$	$Di(n) = R2i(n) + R3i(n)$
N+12	$Di(n)$		$Cr(n+1) = Ar(n+1) + Br(n+1)$
N+13	$Cr(n+1)$	$Ai(n+1)$	$R1r(n+1) = Ar(n+1) - Br(n+1)$
N+14		$Bi(n+1)$	$Ci(n+1) = Ai(n+1) + Bi(n+1)$
N+15	$Ci(n+1)$	$Wr(n+1)$	$R1i(n+1) = Ai(n+1) - Bi(n+1)$
N+16		$Wi(n+1)$	$R2r(n+1) = R1r(n+1) * Wr(n+1)$
N+17			$R3r(n+1) = R1i(n+1) * Wi(n+1)$
N+18			$R2i(n+1) = R1i(n+1) * Wr(n+1)$
N+19			$R3i(n+1) = R1r(n+1) * Wi(n+1)$
N+20		$Ar(n+2)$	$Dr(n+1) = R2r(n+1) - R3r(n+1)$

input bus size = 32 bits;          output bus size = 32 bit;

# of execution steps per data sample = 13

# of overlapped steps in two adjacent data samples = 3

average efficiency of processors = 100%; bus utility = 45%

**TABLE 3.6** Overlap time space diagram of DIF structure 6.

	DIF 1	DIF 2	DIF 3	DIF 4	DIF 5	DIF 6
# of FPU chips(AMD29325) needed	10	6	4	2	1	1
data bus size (bits)	320	128	128	128	128	64
# of executed steps	5	6	6	8	13	13
average efficiency	100%	56%	83%	100%	100%	100%
# of overlap steps	4	3	3	3	3	3
bus utility	100%	83%	83%	50%	25%	45%
total # of executed steps for 1024 real data points	516 *10 =5160	1539 *10 =15390	15390	26530	51230	51240

**TABLE 3.7 Comparison of 6 DIF butterfly structures**

### **C. SOME VHDL BEHAVIORAL MODELS**

The objective of this section is to describe a VHDL modeling effort to verify an FFT system design and show the benefit of VHDL simulation at the data flow level. Only structure 1 mentioned previously is used.

#### **1. FULL PIPELINE DIF BUTTERFLY STRUCTURE**

The structure 1 mentioned in the previous section is a full pipeline structure. Figure 3.7 shows 10 processors and several internal registers holding previous partial results. There are some other registers used to hold weight coefficients and output data produced by this butterfly structure. There are no multiplexed buses for input and output data.

In order to reduce the response time of this butterfly structure, two different triggers are employed. Floating point processors are positive edge triggered. The registers are negative edge triggered. In this way, only three and a half clock periods are needed to complete one butterfly operation. Otherwise, it would require 7 clock periods if either positive or negative edge is employed alone. To avoid undesirable signal data entering into this butterfly structure, and undesirable output data generated out of it, enable signals, IE, OE, and ENABLE are needed. In this structure butterfly, the signal IE is used for input register enable, the signal OE for output register enable, and the signal ENABLE for



processor enable. How to generate those enable signals IE, OE, and ENABLE with appropriate timing is discussed in the following VHDL model.

## **2. CONTROLLER FOR THE BUTTERFLY STRUCTURE**

This controller is designed to produce not only the enable signal for the butterfly but also requests for input to FFT butterfly and output to be stored. Figure 3.13 shows the flow chart of the controller and its logical symbol. The controller communicate with its environment via seven signals, 2 for input and 5 for output. IN\_R is an output signal used for input data request. OUT\_A is an output signal used to show output data available on the output bus. IN\_E is an input signal showing that the input data fetched has been completed. OUT\_E is an input signal showing that the output data has been stored. Both signals IN\_R and OUT\_A are generated by the controller, while signal IN\_E and OUT\_E are produced by the address sequence generator. Signals IE, OE, and ENABLE, which were mentioned in previous section, are generated by this controller which was needed to manipulate the butterfly structure. CNT is an internal counter in this controller. In this thesis, the action of "set a signal" means that a signal is set 1, while "clear a signal" means that a signal is set to 0. The flow chart shows the activities as below:

- Initially, it is triggered by IN\_E and OUT\_E generated by the address sequence generator.

- It will initiate IE and ENABLE to activate the butterfly. It also sets IN\_R, clears OUT\_A, and asks for data fed from RAM into butterfly.
- At the proper time, the output of the butterfly would be available by setting OE. When data becomes available at the output, this controller ask its environment to store the output data by setting OUT\_A.
- When IN\_E is set meaning that the input data is fetched to the end of the input data set, the controller would stop input data fetching by clearing IN\_R, and close the imports of the butterfly by clearing IE.
- Finally, when OUT\_E is set due to finishing the data set, the controller would close the output port of the butterfly immediately by clearing OE, and then clear OUT\_A.

The input data is going to be fed into the butterfly by setting the IN\_R. However, in the above description it did not mention clearly when the output port of the butterfly structure would be open. Table 3.1 shows that 5 steps occur between fetching the data from RAM to producing output on the data bus. The internal counter, CNT, is used to detect the 5th clock period after the controller initiated the butterfly and the first input data was fed into the butterfly. When the number in CNT is 5, the controller would automatically set the OUT\_A to indicate that the output data on output bus is available.

### 3. ADDRESS SEQUENCE GENERATOR

According to Figure 3.5, there is a need to obtain data from memory and feed it to butterfly to achieve the calculation of an eight point FFT. Hence, the main functions

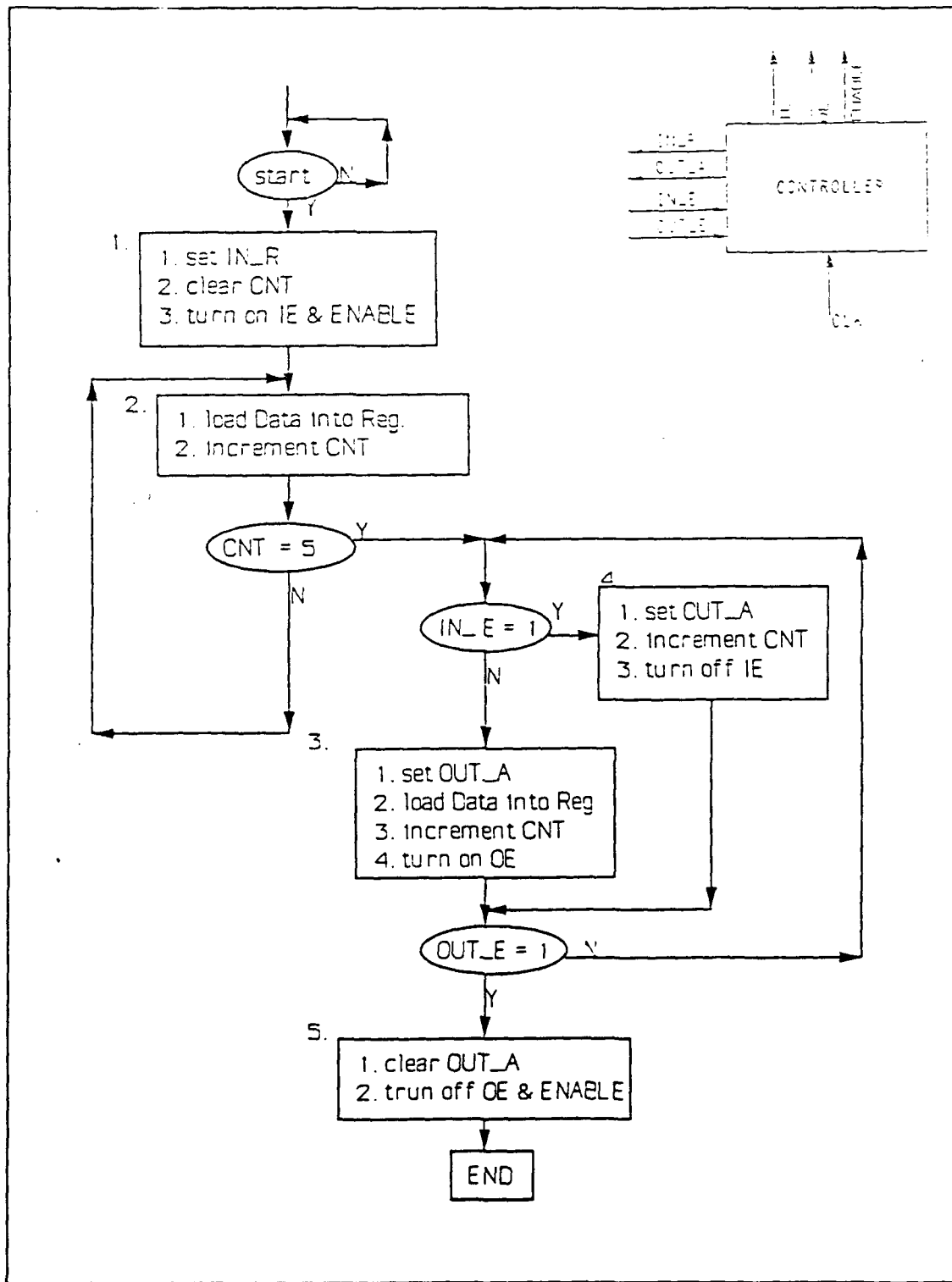


FIGURE 3.13 Controller flow chart and its logical symbol.

of this generator are to produce the input and output addresses for memory access, read/write signals, and memory chip enable signals. In this thesis, the non-bit-reversal algorithm is implemented. The input and output addresses associated with the butterfly are generated according to Figure 3.5. In Figure 3.14, these signals for data bus addresses include ADD1, ADD2, and ADD3. Memory enable signals contain chips enable OE1, OE2, and OE3. Memory read/write signals R1/W1, R2/W2, and R3/W3 are also required. Since it is necessary to fetch input data A, B,  $W^k$  concurrently, three RAM modules are used. The signal OE3, R3/W3, ADDR3 are used to fetch the weight coefficient  $W^k$  from RAM. Signals OE2, R2/W2, and ADDR2 and signals OE1, R1/W1, and ADDR1 are used to access memory RAM 0 and RAM 1 respectively. The connection of RAM and butterfly is shown in Figure 3.17. ADD1, ADD2, and ADD3 are shown with bold signal lines representing a bus.

Another function of this generator is to cooperate with the controller. They cooperate via four signals IN\_E, OUT\_E, IN\_R, and OUT\_A which were mentioned in the previous section. Figure 3.15 is the flow chart of the address sequence generator. State 5 and state 6 of Figure 3.15 occur when the predetermined  $2N$  value has been reached.  $2N$  is the number of data samples of the FFT. The address sequence generator also cooperates with the universal controller at a higher level of hierarchy. The interface includes input signals CHE, LEN, and ISTO, and output signals STAGE\_CNT, OSTO and FFT\_CMP. CHE

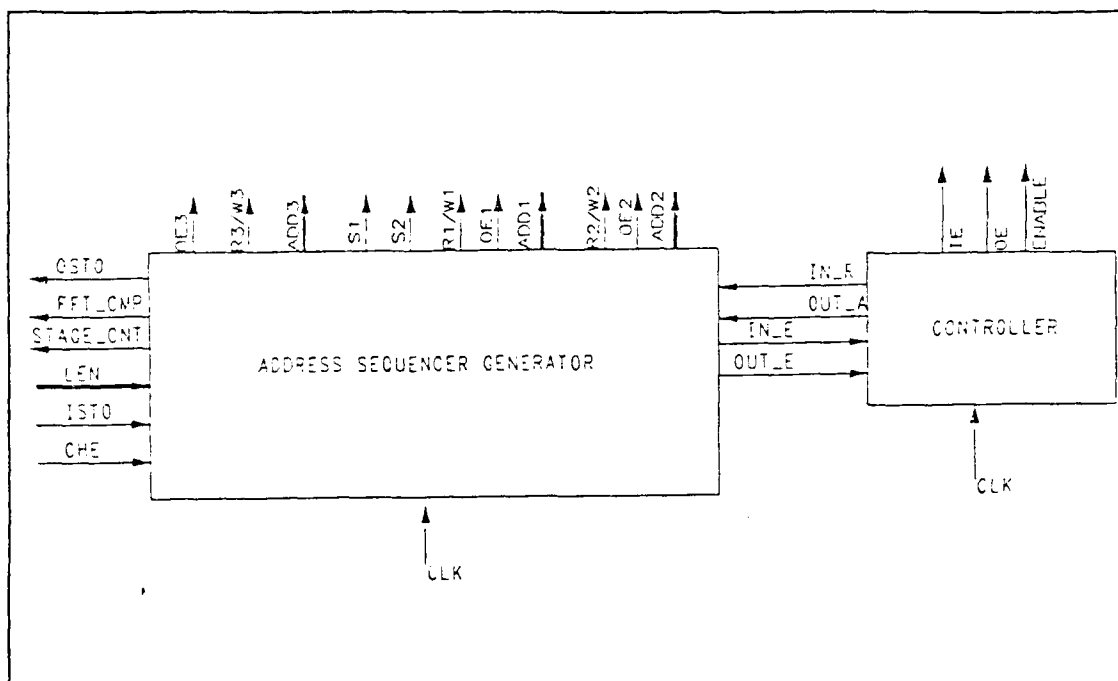


FIGURE 3.14 The block diagram of address sequence generator and controller.

represents chip enable. LEN represents the input data length. ISTO represents a pointer signal of the initial input data in the RAM. STAGE\_CNT represents stage counter number in the FFT algorithm. OSTO represents a pointer signal of the output data in the RAM. FFT\_CMP represents the FFT completion. Before the beginning of the FFT data flow, the universal controller loads N number of pairs of input data, and sets N on the signal LEN. It uses the signal ISTO to indicate which of the two RAM, RAM 0 or RAM 1, the input data is stored. For example, in Figure 3.18 if the input data is stored in RAM 1, the signal ISTO would be set to 1. The universal controller uses signal CHE to start the address sequence generator. The signal STAGE\_CNT keeps a number to tell the external universal controller which

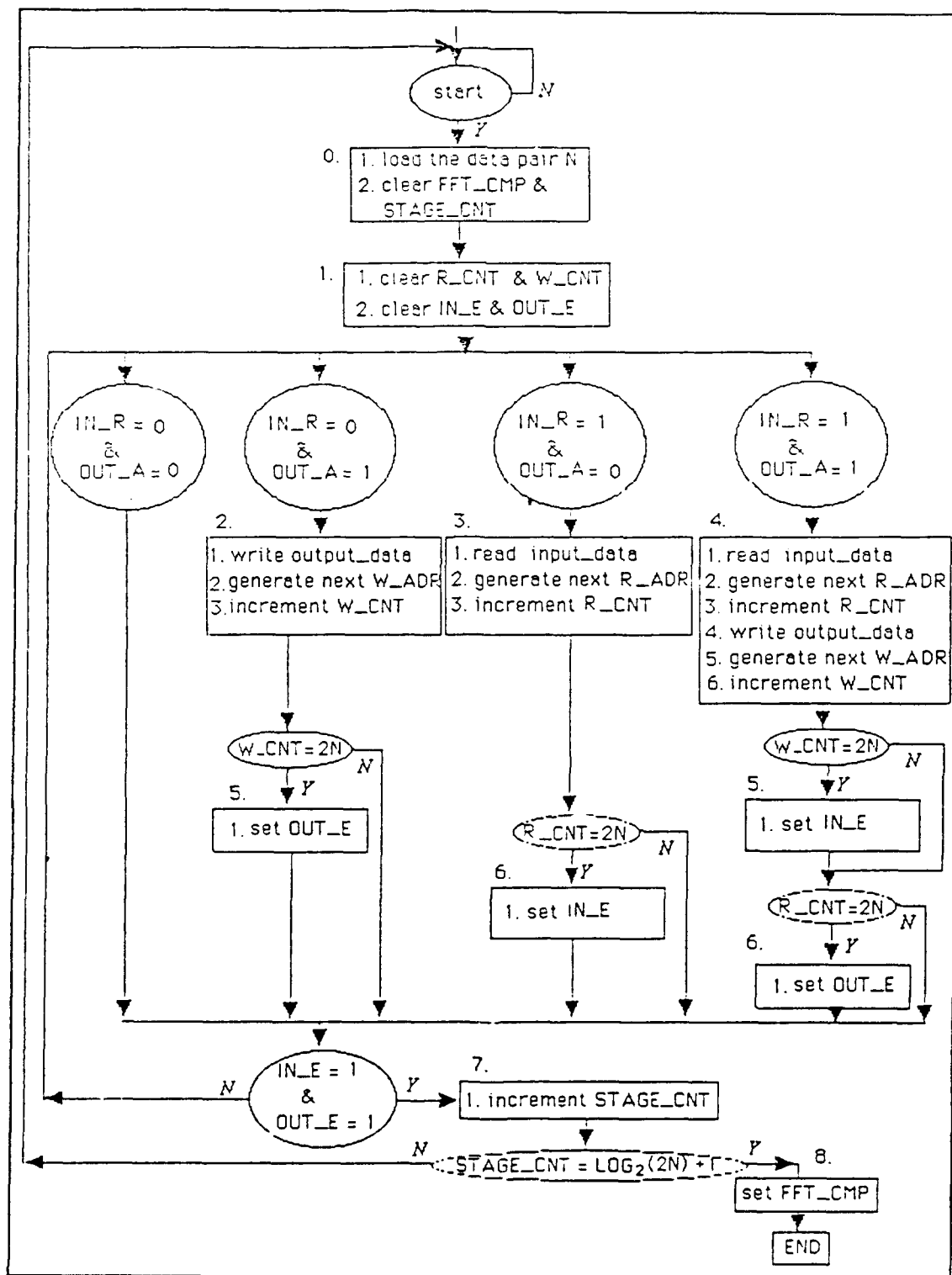


FIGURE 3.15 Address sequence generator flow chart.

stage of the FFT is executed currently in the butterfly. For example, if the number of pairs of data to the FFT is 4, which is an 8-point FFT shown in Figure 3.5, the total executable stage is 3, which results from the  $\log_2(8)$ . The number in the STAGE\_CNT would count from 0 to 2. As shown in Figure 3.15, once the signal STAGE\_CNT reaches 3, the signal FFT\_CMP would be set. This represents the FFT completion. The signal OSTO is used to indicate where the output data is available from the two RAMs.

Selection signals S1 and S2 are used to control the "3 to 1" selector, shown in Figure 3.18. There is another way for memory access to provide data to the universal controller. Before the universal controller starts the FFT, it would store the input data set into one of the two RAMs using those memory access signals drawn at the bottom of Figure 3.17 and 3.18. Those signals drawn at the bottom of Figure 3.17 and 3.18 include the signals of memory access OCH1, OCH2, OCH3, OR1/OW1, OR2/OW2, OR3/OW3, CADD1, CADD2, and CADD3, selection signal C1, C2, and output enable BE. Those signals provide a way that the universal controller can use to fetch input data and store the results of the FFT. For example, for a complete 8-point FFT, which are initially stored in RAM 1 shown in Figure 3.18, the universal controller would set N to 4 on signal LEN and use selection signals C1, C2, and one group of memory access signals OCH 1, CADD 1, and OR1/OW1. It will indicate where the input data is stored by setting ISTO to 1.

Then it activates the address sequence generator by setting CHE. In the execution process of the FFT, the signal STAGE\_CNT would tell the universal controller which stage of FFT is active. Using S1 and S2 and two groups of memory access signals, the address sequence generator selects the input data from RAM, and stores the output data of the FFT butterfly back to RAM. When the FFT is done, the address sequence generator responds to the universal controller by setting signal FFT\_CMP. Signal OSTO, in this case being 0 at the end of the FFT, would indicate where the results of the FFT are stored. According to the pointer OSTO, the universal controller would fetch the results of the FFT from RAM 0 via CADD2, R2/OW2, OCH2 and BE.

In the following, the activities of the address sequence generator can be summarized. Let R\_CNT and W\_CNT be the internal counters of read and write operations. The source program of the address sequence generator and the controller are attached in Appendix E.

- First, clear FFT\_CMP and STAGE\_CNT. Load N with predetermined number of pairs of data to be transformed.
- Second, clear R\_CNT, W\_CNT, IN\_E and OUT\_E.
- Third, check the status of IN\_R and OUT\_A generated by the controller in the following.
  1. When both IN\_R and OUT\_R are clear, the controller is not ready, so the address sequence generator would wait until IN\_R is set.
  2. When the IN\_R is set, the controller is ready, and the butterfly needs to be fed with data. The number stored in R\_CNT is incremented by 1.



3. When OUT\_A is set, the controller had opened the output port of the butterfly, and the data on the output data bus is available. The number stored in W\_CNT is incremented by 1.

4. When both IN\_R and OUT\_A are set, the butterfly needs to be fed with data, and the output data coming from it are available on the output data bus. The number stored in R\_CNT and W\_CNT are incremented by 1.

- Fourth, check the number of R\_CNT and W\_CNT, if the predetermined number is reached for each counter, the stop signals IN\_E and OUT\_E would be transmitted to the controller. For example, when the data read is complete, the IN\_E would be set.
- Finally. Once the IN\_E and OUT\_E are set. The address sequence generator would increase the STAGE\_CNT and compare it with the total stage number required. If they are not yet the same do the next stage again. For example if the total number of pairs of data is 4, the execution stages should be 3. If the number in the STAGE\_CNT has counted to this execution stage number, the address sequence generator would set the signal FFT\_CMP, indicating that the FFT operation is completed.

#### 4. RAM

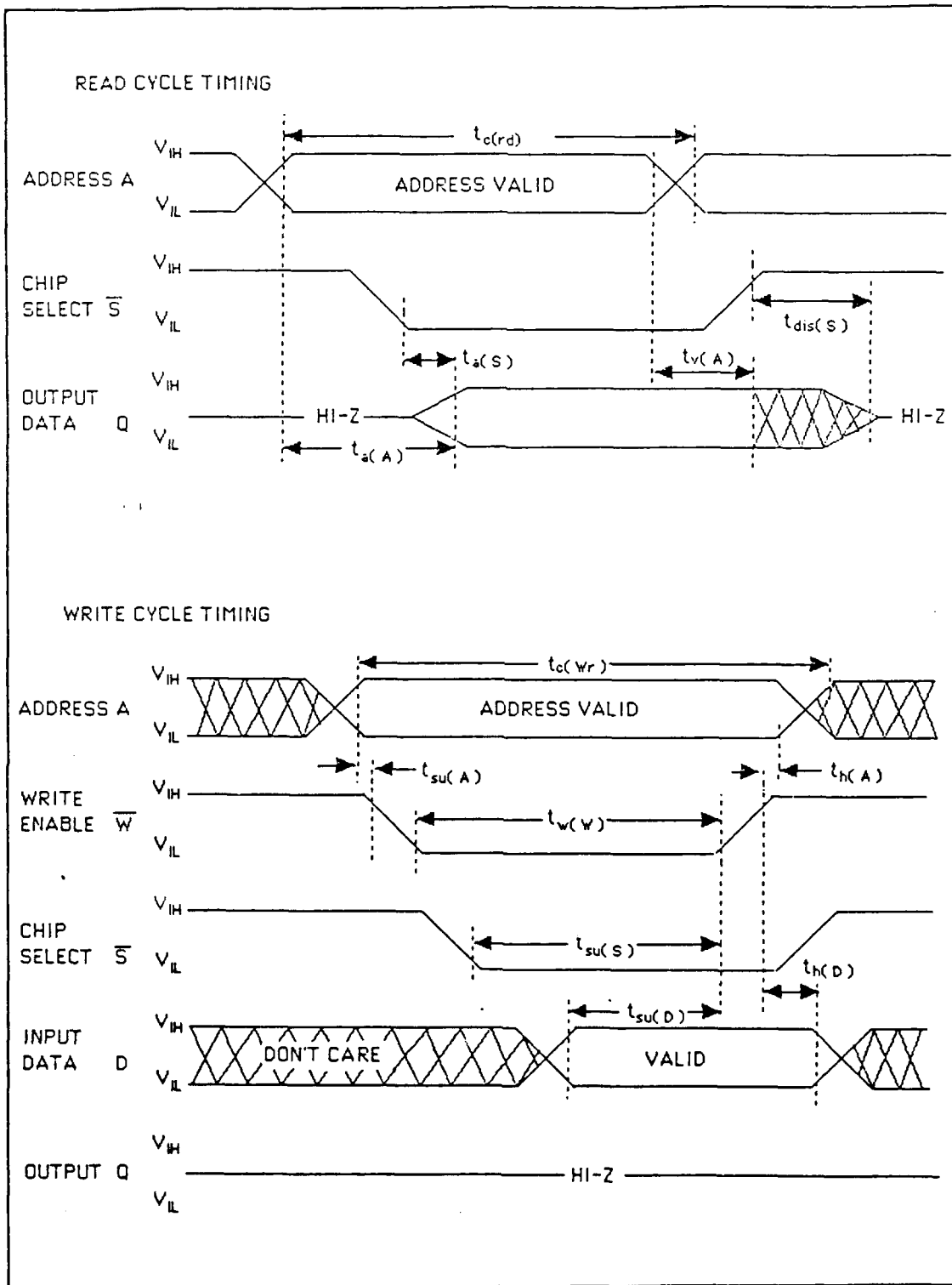
Since there is memory storage required in this structure, a random access memory model is necessary for the VHDL simulation. In order to reduce the complexity of the signal timing in RAM and simplify the model of the RAM, only static RAM, having a separate input and output data bus was implemented. The size of the RAM is 256 by 32, because input is a 32-bit floating point number. Several parameters, for example, data set up time and access time associated with the read cycle and the write cycle are shown in Figure 3.16. The RAM VHDL model is attached in Appendix F. As mention above,

only a few timings are concerned in this model program. If someone needs a larger sized RAM, he can change the size of the local variable DATA\_MATRIX to increase the storage of the RAM.

#### **D. SIMULATION OF THE DATA FLOW DESIGN OF FFT**

Right now, several VHDL models which are associated with the data flow of the FFT system were built. In order to reduce the total size of the FFT design, and have a faster simulation, several elements are left out, The 2 to 1 selectors, registers, and buffers were not modeled at the chip level. Their behavior is described in the data flow design of the FFT for simulation.

Shown in Figure 3.17 is an original description, where 6 pairs of RAMs with 256 by 32 bits are required to read and write data. Three 2 to 1 selectors are used to decide where input data is to be fetched from and where output data is to be stored. In Figure 3.17, the universal controller uses signal C1 and memory access signals of RAM 1 or RAM 2 to select data on the input bus and store it into RAM 1 or RAM 2 respectively. In this situation, each RAM module contains three blocks of RAM for storing A, B, and coefficient  $W^k$ . Assuming that the initial input data is stored in RAM 1, the universal controller would load the length of the input data pairs on signal LEN. It then indicates where the input data is by setting signal ISTO. The universal controller also uses CHE



**FIGURE 3.16** Timing of read cycle and write cycle (adopted from National CMOS RAM data book).

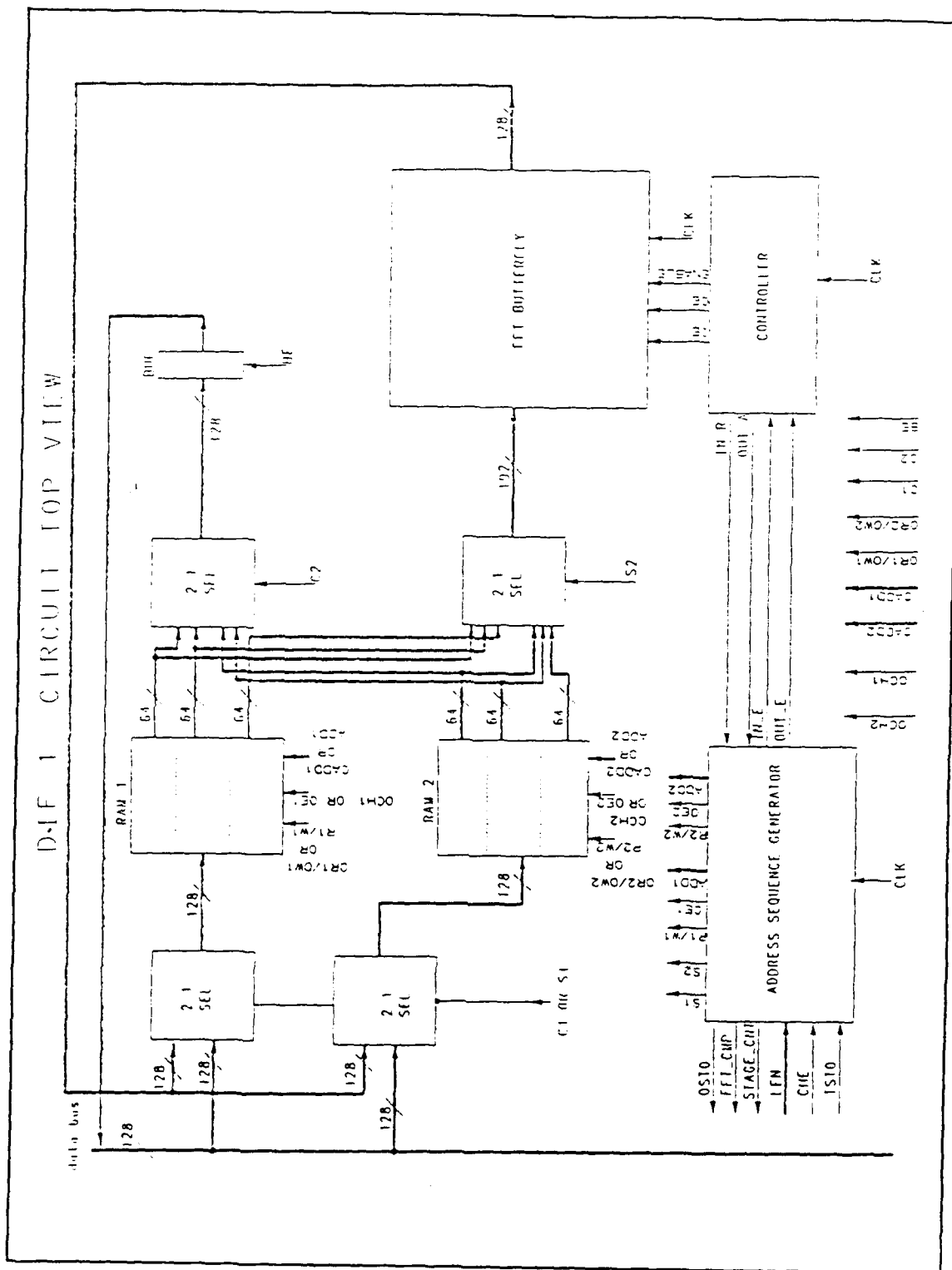


FIGURE 3.17 The original data flow system of FFT.

to trigger the address sequence generator. The address sequence generator would generate access signals OE1, R1/W1, and ADD1 to fetch the first input data to the FFT butterfly after the controller has been initiated by the signal IN\_E and OUT\_A. Since the universal controller stores the input data in RAM 1, it will store output data from the butterfly of the first stage to RAM 2 via the selector enable S1. As shown in Figure 3.5, the output data of the first stage would then be of the input data of the second stage. The output data of the second stage FFT would again be stored back to RAM 1, and so on and so forth. If the input data number is 8, as shown in Figure 3.5, the total number of execution stages is 3. In the manipulation of the data flow, the signal STAGE\_CNT always reveals to the universal controller which stage is being executed. At the end of the FFT operation, the address sequence generator would indicate to the universal controller about where the final output data is stored via the pointer signal OSTO. The completion flag is then set on the signal FFT\_CMP.

Since the original FFT design in Figure 3.17 is too large to be accommodated in the VAX VMS 4.5 operating system, the revised version of the design is created in Figure 3.18. In Figure 3.18, all the data flow operations are similar to what was mentioned earlier with the exception of the number of selectors, RAM size, and internal data buses used are reduced. The size of the internal bus lines was reduced from 128 to 64.

In Figure 3.18, the output data bus of the FFT butterfly contains C and D outputs. It is split into two separable data buses of size 64 and multiplexed into RAM. The two registers A and B shown in Figure 3.17 are triggered at different edges of the clock, because the output data of RAM with size 64 can not convey two complex numbers, which requires a size of 128. The complex data, therefore, needs to be multiplexed onto the two registers. This design was successfully simulated on the VMS 4.5 operating system. In Table 3.8, a successful example of the simulation result of the revised FFT system is shown. The flow chart of the universal controller is shown in Figure 3.19.

In this chapter the data flow models of a FFT system was discussed. This is a full pipeline structure that requires several VHDL models. In the next chapter, using of the created FFT system for a Discrete Cosine Transform is discussed.

input data have 8 complex number

-2.0 - 1.0j , 2.0 + 1.0j  
-3.0 + 2.0j , 1.0 - 2.0j  
4.0 - 2.0j , 1.0 - 5.0j  
3.0 - 2.0j , 3.0 + 1.0j

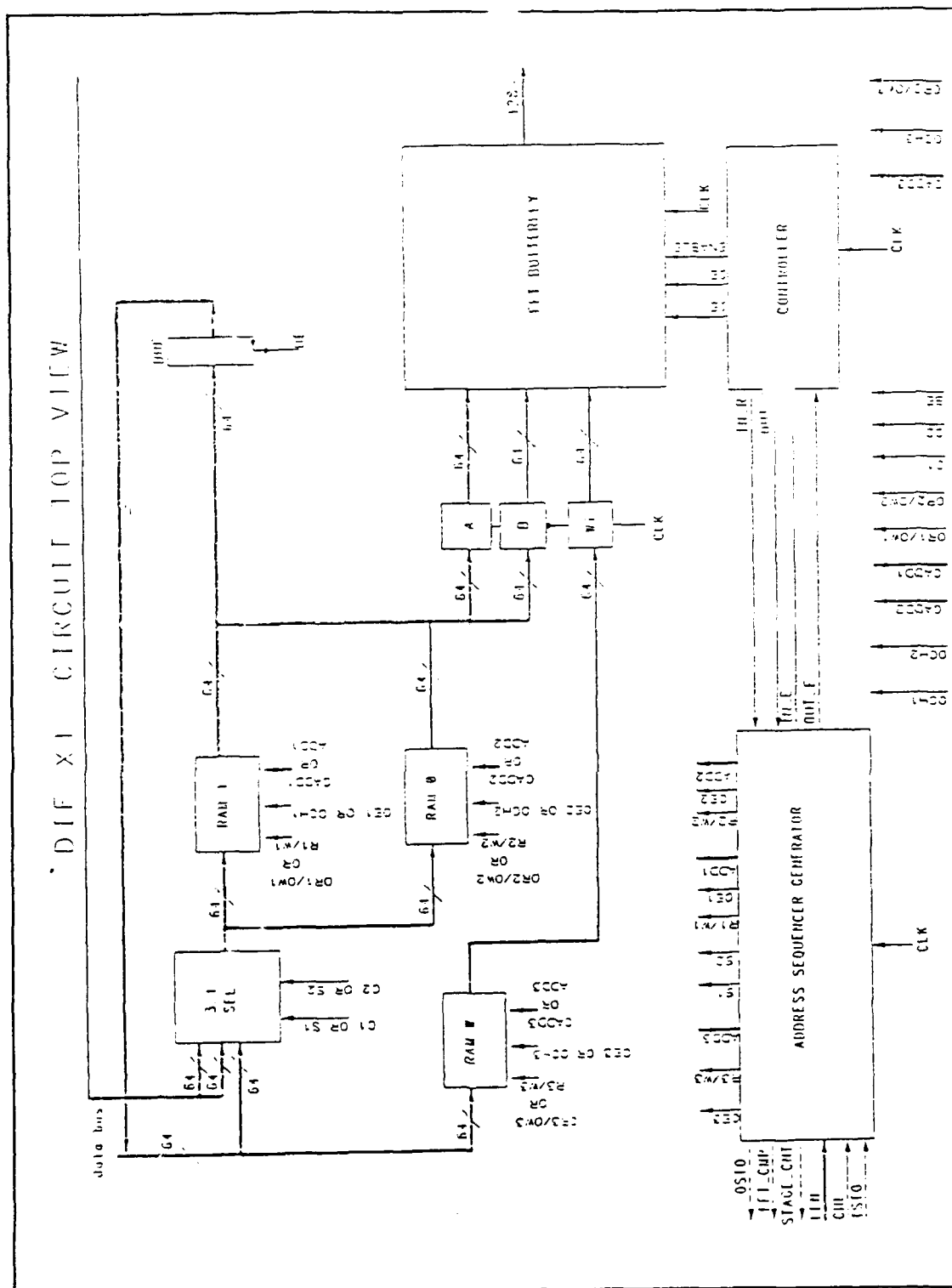
output data using MATLAB function

9.0 - 8.0j  
2.2426407 + 14.0710678j  
- 1.0 - 2.0j  
-10.0 - 10.6568542j  
- 5.0 + 2.0j  
-6.2426407 - 0.0710678j  
5.0 - 4.0j  
-10.0 + 0.6568542j

output data using simulated program

9.0 - 8.0j  
2.2426407 + 14.0710677j  
-1.0 - 2.0j  
-10.0 - 10.6568542j  
-5.0 + 2.0j  
-6.2426407 - 0.0710602j  
5.0 - 4.0j  
-10.0 + 0.6568532j

**TABLE 3.8 Comparison of the FFT result of using the MATLAB function and this simulated FFT system.**



**FIGURE 3.18** The revised data flow system of FFT.



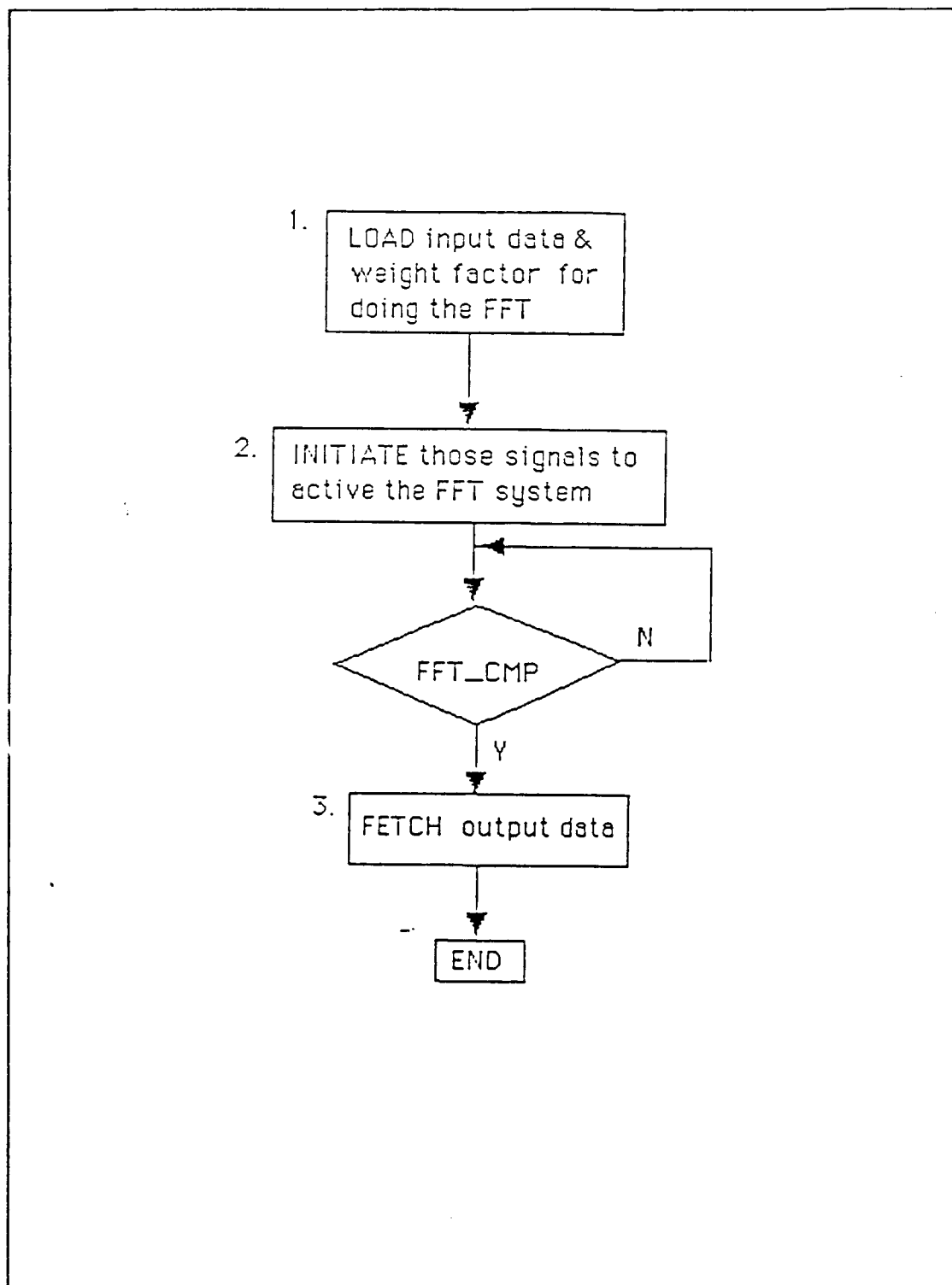


FIGURE 3.19 The flow chart of the universal controller.

#### IV. THE DATA FLOW DESIGN OF THE DISCRETE COSINE TRANSFORM

##### A. INTRODUCTION TO DISCRETE COSINE TRANSFORM(DCT)

In the previous chapter, the Fast Fourier Transform implementation was discussed. In this chapter, the discussion is focused on the DCT using the system designed for FFT. Applications of the DCT include image data compression, coding, and storage.

Before the structure of DCT system is designed, it is necessary to know the difference between the formula of Discrete Cosine Transform, and the formula of Fast Fourier Transform. The one-dimensional DCT for a limited sequence  $\{u(n), 0 \leq n \leq N-1\}$  is defined as

$$V(K) = \alpha(K) \sum_{n=0}^{N-1} u(n) \cos(\pi(2n+1)k/2N) \quad (4.1)$$

$$\alpha(0) = \sqrt{1/N} \quad \text{for } K=0 \quad (4.2)$$

$$\alpha(K) = \sqrt{2/N} \quad \text{for } K=1 \dots N-1 \quad (4.3)$$

From the equation (4.1), the relationship between DCT and FFT is derived as,

$$V(K) = \text{Re} [ \alpha(K) e^{-j2\pi k/2N} * U(K) ] \quad (4.4)$$

$$U(K) = \sum_{n=0}^{N-1} u(n) e^{-j2\pi kn/N} \quad (4.5)$$

The total number of input sequence  $N$  must be an integer number of power of 2 [Ref. 9]. From the equation (4.4) conversion of the FFT to the DCT can be done in 3 steps, a complex multiplication, a scale multiplication, and taking the real part of the data. This requires two real multiplication, one addition, and one scale multiplication when floating point operations are counted.

The scale factor  $\alpha(K)$  and the FFT weight factor  $W^{k/2}$  can be merged, which can be written as

$$H^{k/2}(k) = \alpha(K) * W^{K/2} \quad (4.6)$$

In this way, it is possible to reduce the number of multiplications from 3 to 2. Prior to calculating the DCT, the data from the FFT calculation and scale weight factor  $H^{k/2}(K)$  must be stored in RAM. Then, two real data multiplications and one addition will yield the result.

## B. THE DISCRETE COSINE TRANSFORM SYSTEM IMPLEMENTATION

Two methods to implement a DCT system are discussed here. One is to use the full pipeline structure, the other is to modify the universal controller of the FFT system discussed in the previous chapter.

In Figure 4.1, a full pipeline structure uses 3 additional processors, 2 for multiplication and 1 for addition. In other words, once the output data from the FFT system is stored in memory, additional circuitry is used to perform two multiplications and one addition to obtain the Discrete Cosine Transform. In addition, this requires the memory address sequence generator to access data stored in RAM.

Figure 4.2 shows the block diagram of the FFT and the external universal controller. The interface signals include three groups signals. The first group of signals shown at the bottom of the Figure 3.18, C1, C2, OR1/OW1, OR2/OW2, OR3/OW3, OCH1, OCH2, OCH3, CADD1, CADD2, CADD3 and BE, are associated with memory access in the FFT system. The second group of signals, shown at the lower hand corner in Figure 3.18, include LEN, CHE, and ISTO which are used to initiate the address sequence generator in the FFT system. The third group of signals, OSTO, FFT\_CMP, and STAGE\_CNT, are the status signals from the FFT system.

A second method of implementing the DCT is shown in Figure 4.3. The universal controller discussed in the previous

chapter is modified to complete the Discrete Cosine Transform of the input data. In the Figure 3.3, the butterfly structure of DIF non-bit-reversal algorithm was shown where the input and output have the following relationship.

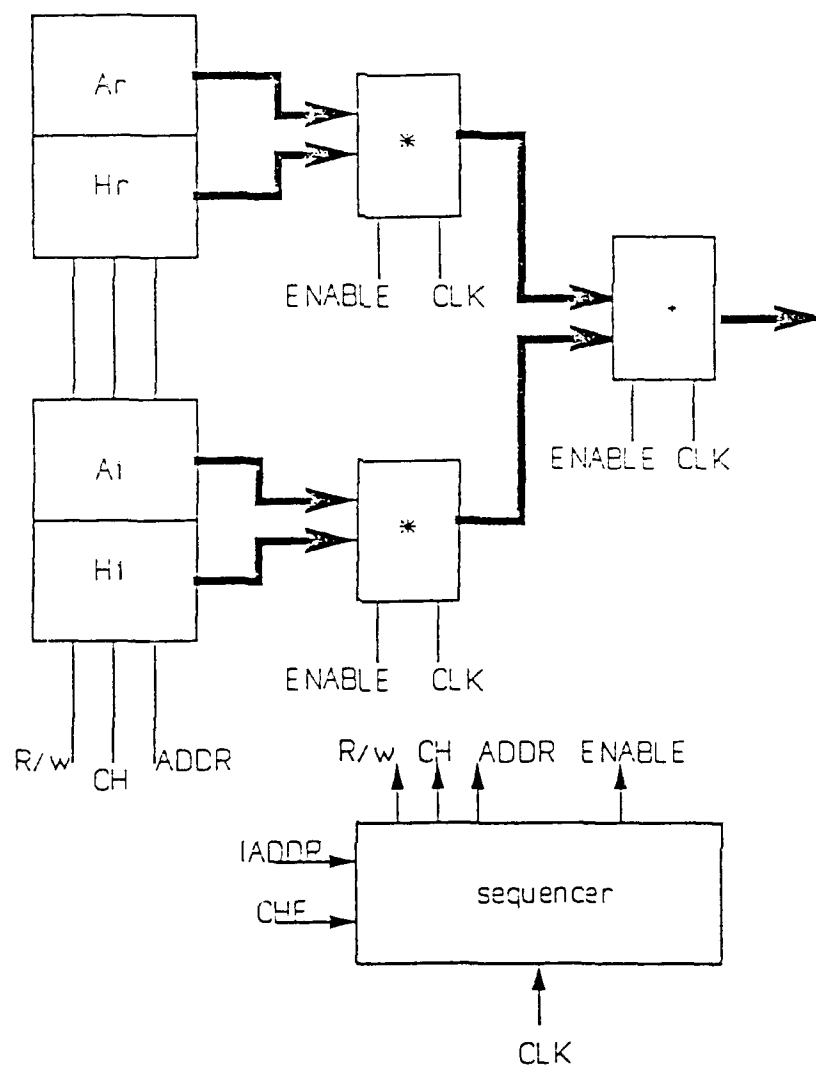
$$C = A + B \quad (4.7)$$

$$D = (A - B) * W^k \quad (4.8)$$

A, B, and  $W^k$  are input data, whereas C and D are output data. Based on equation (4.7) and equation (4.8), let  $W^k$  be  $\alpha(K) * e^{-j\pi k/2N}$ , A be  $U(K)$ , and B be 0. In this way the same butterfly can yield another output D. For Discrete Cosine Transform, only the real part of D is kept. After the complete output data of FFT is generated, the result of DCT is needed to go through the butterfly for one more cycle. The real part of the output data is the result of the Discrete Cosine Transform. It is straight forward to modify the flow chart of the universal controller of Figure 3.19. After the complete output data is generated from the FFT butterfly, one more cycle through the butterfly is needed if we want to do DCT for original input data.

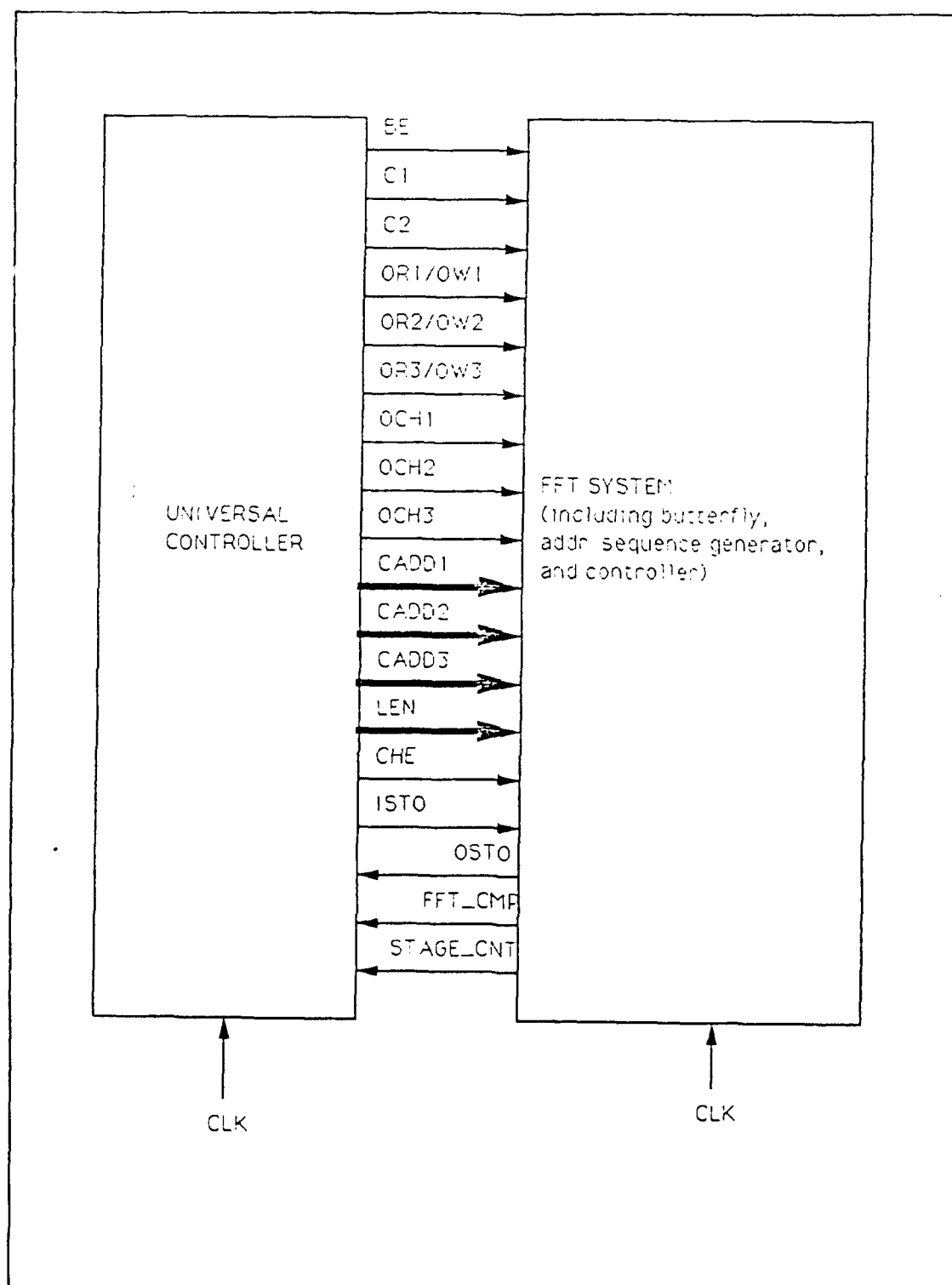
If the first method is used, it is necessary to build additional circuitry, with 3 processors and a local memory access sequence generator. If it is undesirable to build any additional circuitry, method two can be adopted. This approach will complicate the universal controller. Therefore, there is a trade off between these two methods.

The idea of how to get a Discrete Cosine Transform result using an FFT structure is discussed here. In the next chapter, the improvement and future research of this thesis will be discussed



Ar: the real part of data coming from FFT system output  
 Ai: the imaginary part of data coming from FFT system output  
 Hr: the real part of scale weight factors  
 Hi: the imaginary part of scale weight factors  
 ADDR, R/W, & CH are the memory access signal  
 IADDR is the initial input data address  
 CHE is the chip enable of sequencer

**FIGURE 4.1 Full pipeline structure to implement the DCT system, the input data come from the FFT system output.**



**FIGURE 4.2** Block diagram of the universal controller and the FFT system.



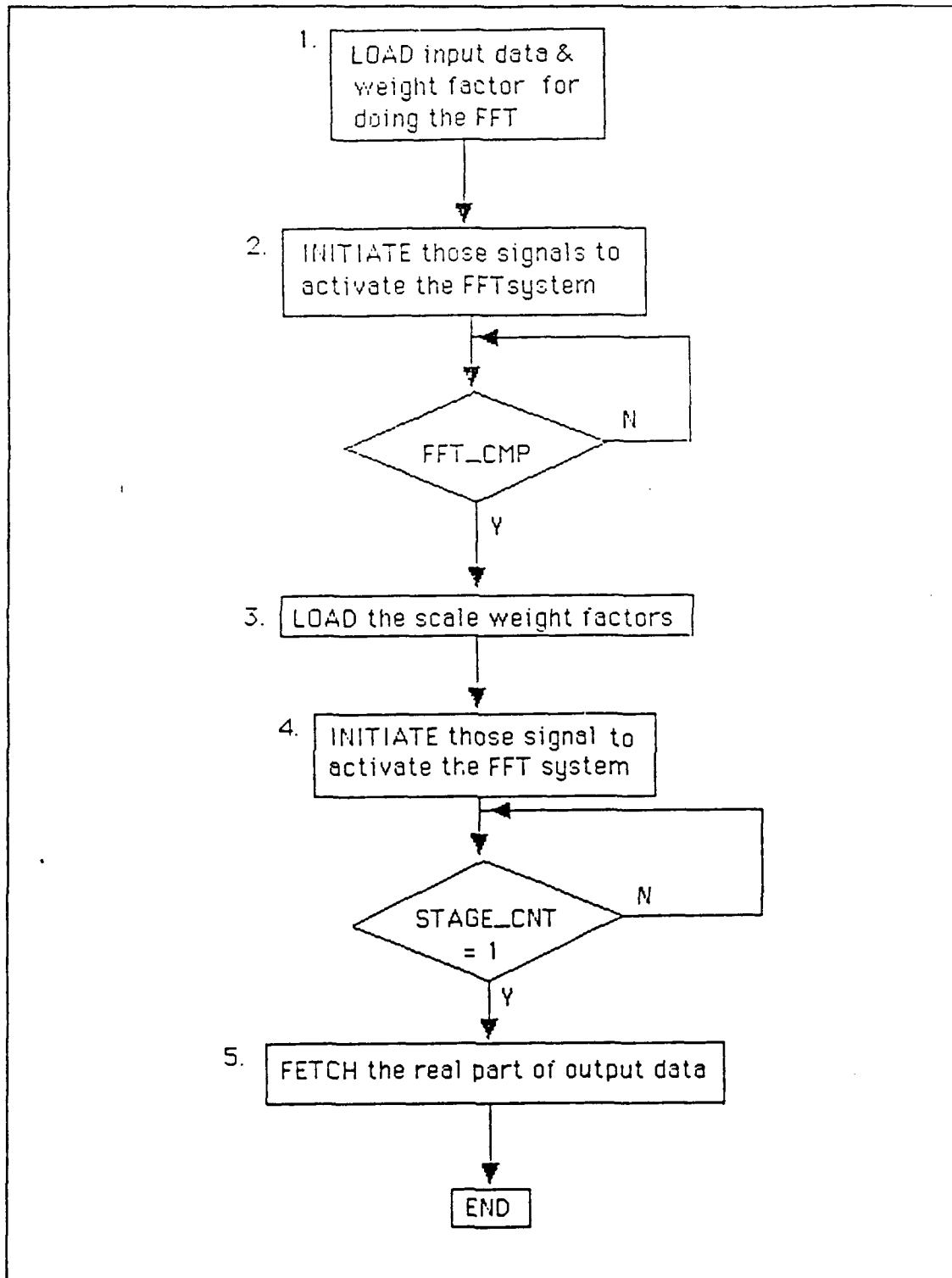


FIGURE 4.3 Modified flow chart of the universal controller.

## V. CONCLUSION

### A. CONCLUSION

Although this thesis modeled the floating point arithmetic processor "AMD29325", data flow FFT systems, and the DCT system, the methodology can be applied to other digital signal processing systems. Many signal processing algorithms require sum-of-product operations that are well suited to designs discussed in this thesis.

In this thesis, the data flow design of FFT in the full pipeline butterfly structure has been built and the model has been verified. The result is shown in the Table 3.8. Due to limitation of time the data flow design of DCT is not fully simulated. Many problems had been encountered in the study. A few problems were easy to solve such as the syntax errors, but many problems were difficult to overcome. A "trial and error" approach was often taken. There are still unresolved problems. One problem is related to the source programs created under VHDL version 1.5 that can not run under VHDL version 2.0. This problem developed due to the software version change. In the Intermatrix VHDL version 1.5, there are several internal problems. For example, it can not print a negative value in the report file. It can not generate a triggered pulse waveform in the interactive simulation mode. When the "BLOCK"

is used in the VHDL source program, it would generate some unexpected side effects.

The very important experience here is how to deal with system design in top-down design methodology and how to use VHDL simulation to analyze systems to get an optimum design. Hierarchical design is an important approach that allows step by step solution to circuit design.

#### **B. IMPROVEMENTS AND FUTURE RESEARCH**

The data flow designs of a Radix 2 FFT in DIF algorithm and the data flow designs of a DCT had been discussed and implemented in this thesis. However, several areas in this thesis can be improved. For example, in Chapter III the original FFT design does not run on the VMS 4.5 operating system because of the size and complexity of the design used in the source program. It is replaced by the revised program which is shown in Figure 3.18. In Table 3.8 there are still some errors in rows 1, 6, and 8 of the output data from the FFT system simulation. These errors were caused by truncation. In this thesis truncation was used to deal with the large values generated when the length of mantissa size exceeded 23 bit of the IEEE mantissa size pattern. For further improvement a rounding method should be used. Several directions are listed in the following for future research.

### **1. TO IMPLEMENT THREE ADDITIONAL PRECISION FORMATS TO IMPROVE THE ARITHMETIC ACCURACY**

Only single precision is employed in this thesis. There are three other precision formats: single extended precision, double precision, and double extended precision. These formats are shown in Figure 2.2.

### **2. TO ADD SEVERAL OTHER FUNCTIONS ASSOCIATED WITH THE AMD29325 OPERATION**

In this thesis, only four floating point arithmetic operations are implemented. There are other functions shown in Figure 2.5 associated with the AMD29325 operation including the floating-point constant subtraction, integer to floating-point conversion, floating-point to integer conversion, IEEE to DEC format conversion, and DEC to IEEE format conversion.

### **3. TO PERFORM THE RADIX 4 FAST FOURIER TRANSFORM IN DIT OR DIF ALGORITHMS**

It is possible to further reduce the number of calculations required to perform the FFT by using a radix 4 algorithm provided that the number of input data is an integer power of 4. Two basic signal data flows in DIT and DIF algorithm for radix 4 are shown in Figure 5.1. As shown in Table 5.1, the advantage of the radix 4 algorithm is to reduce the number of multiplications by 25% [Ref. 10].

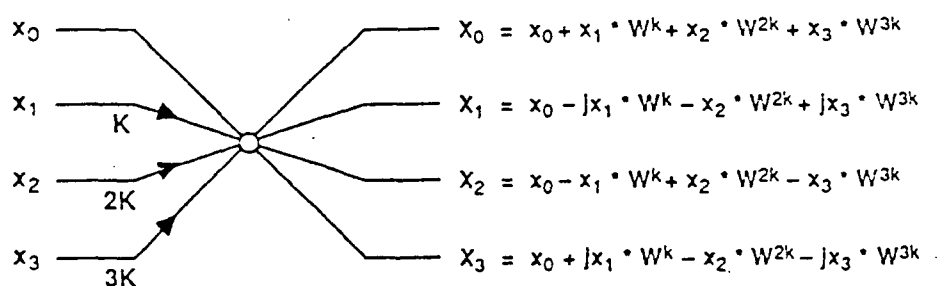
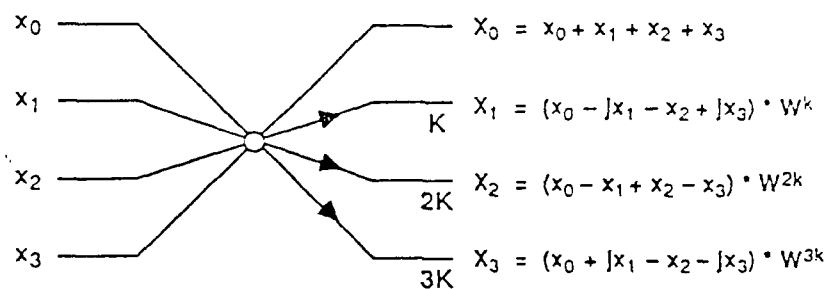


FIGURE 5.1 Butterfly in Radix 4, top is the DIT algorithm, bottom is the DIF algorithm.

Radix 2			Radix 4	
N	(*)	(+)	(*)	(+)
64	192	384	144	384
256	1024	2048	768	2048
1024	5120	10240	3840	10240

TABLE 5.1 The comparison of total number of arithmetic operations needed in Radix 2 and Radix 4.

#### **4. TO IMPROVE THE ADDRESSING SEQUENCE GENERATOR TO REDUCE FETCHING IDENTICAL WEIGHT FACTORS**

In Figure 3.5, the total number of weight factors needed for an 8-point fast fourier transform is 12. The number of fetches for the weight factor is also 12. In fact, only 4 weight factors are different, i.e.  $k = 0, 1/4, 1/2, \text{ and } 3/4$ . If the address sequence generator is modified to recognize the identical weight factors, the memory needed to stored weight factors can be reduced.

#### **5. TO BUILD THE FAST FOURIER TRANSFORM USING A SPECIAL "COMPLEX VECTOR PROCESSOR (CVP)" CHIP**

In order to increase the speed of the FFT simulation program, one special chip for FFT operation called "CVP" [Ref. 11] can be used. The CVP implements a full 32 bit complex multiplication on chip in a single clock cycle. In addition it provides four 40 bit programmable complex accumulators to facilitate operations in radix-2 and radix-4 algorithms.

## APPENDIX A: THE ELEMENT FUNCTIONS OF THE FPU

--these element functions associated with FPU(floating point unit)

```
library std ;
use std.standard.all;
package refer is
type BIT_ARRAY is array( integer range<> ) of BIT;
type BIT_MATRIX is array( integer range<> ) of BIT_ARRAY(31
downto 0) ;
type FLAG is
  record
    ovf_bit:BIT;
    unf_bit:BIT;
    nan_bit:BIT;
    zero_bit:BIT;
  end record;
type LOGIC_LEVEL is ('1','0','X','Z');
type LOGIC_ARRAY is array( integer range<> ) of LOGIC_LEVEL ;
type LOGIC_MATRIX is array( integer range<> ) of LOGIC_ARRAY(
31 downto 0) ;
constant d_precision: integer := 64;
constant s_precision: integer := 32;

function BITSARRAY_TO_FP( bits: BIT_ARRAY)
  return REAL ;

function FP_TO_BITSARRAY( fp: REAL; length: NATURAL)
  return BIT_ARRAY ;

function INT_TO_BITSARRAY( int,length: NATURAL)
  return BIT_ARRAY;

function BITSARRAY_TO_INT( bits: BIT_ARRAY)
  return NATURAL;

function UNHIDDEN_BIT( bits: BIT_ARRAY)
  return BIT_ARRAY;

function SHIFL_TO_R( bits: BIT_ARRAY ; times :integer)
  return BIT_ARRAY;

function IS_OVERFLOW( exp_bits: BIT_ARRAY;
  precision:INTEGER)
```



```

    return BOOLEAN;

function IS_UNERFLOW( exp_bits: BIT_ARRAY;
                      precision: INTEGER)
    return BOOLEAN;

function IS_ZERO( bits: BIT_ARRAY)
    return BOOLEAN;

function IS_NAN( exp_bits: BIT_ARRAY)
    return BOOLEAN;

function BECOME_ZERO( bits: BIT_ARRAY)
    return BIT_ARRAY;

function BECOME_NAN( bits: BIT_ARRAY)
    return BIT_ARRAY;

function SET_FLAG( bits, exp_bits: BIT_ARRAY;
                  precision: INTEGER)
    return FLAG;

function ADD(sign_a:BIT; bits_a: BIT_ARRAY; sign_b:BIT;
            bits_b: BIT_ARRAY)
    return REAL;

function INCREASEMENT(bits:BIT_ARRAY; precision:INTEGER)
    return BIT_ARRAY;

function DECREASEMENT(bits:BIT_ARRAY; precision:INTEGER)
    return BIT_ARRAY ;

function BACK_TO_BITSARRAY(exp_bits:BIT_ARRAY;
                          fp:REAL; precision:INTEGER)
    return BIT_ARRAY;

end refer ;

package body refer is

function BITSARRAY_TO_FP( bits:BIT_ARRAY)
    return REAL is
    variable result :REAL := 0.0;
    variable index :REAL := 0.5;
begin
    for i in bits'range loop
        if bits(i) = '1' then
            result := result + index ;
        end if ;
        index := index*0.5;  ---- .5 = 2**(-1)
    end loop;
end BITSARRAY_TO_FP;

```

```

        end loop;
        return result;
end BITSARRAY_TO_FP;

```

```

function FP_TO_BITSARRAY( fp: REAL; length: NATURAL)
return BIT_ARRAY is
variable local: REAL;
variable result: BIT_ARRAY( length-1 downto 0);
begin
    local := fp ;
    for i in result'range loop
        local := local*2.0 ;
        if local >= 1.0 then
            local := local-1.0;
            result(i) := '1';
        else
            result(i) := '0';
        end if ;
    end loop ;
    return result ;
end FP_TO_BITSARRAY ;

```

```

function INT_TO_BITSARRAY( int,length: NATURAL)
return BIT_ARRAY is
variable digit:NATURAL := 2**(length-1);
variable local:NATURAL ;
variable result:BIT_ARRAY(length-1 downto 0);
begin
    local := int ;
    for i in result'range loop
        if local/digit >= 1 then
            result(i) := '1';
            local := local - digit;
        else
            result(i) := '0';
        end if;
        digit := digit/2;
    end loop;
    return result;
end INT_TO_BITSARRAY;

```

```

function BITSARRAY_TO_INT( bits: BIT_ARRAY)
return NATURAL is
variable result :NATURAL := 0;
begin
    for i in bits'range loop
        result := result*2;
        if bits(i) = '1' then

```

```

        result := result + 1;
    end if;
end loop ;
return result ;
end BITSARRAY_TO_INT;

function UNHIDDEN_BIT( bits: BIT_ARRAY)
return BIT_ARRAY is
variable result : BIT_ARRAY(bits'length downto 0);
begin
    for i in bits'range loop
        result(i) := bits(i);
    end loop;
    result(bits'length) := '1'; ----IEEE format
    return result;
end UNHIDDEN_BIT;

function SHIFL_TO_R( bits: BIT_ARRAY; times :integer)
return BIT_ARRAY is
variable number:integer := times;
variable result : BIT_ARRAY(bits'length-1 downto 0);
begin
    for i in bits'range loop
        result(i) := '0';
    end loop;
    while number <= bits'length-1 loop
        result(number-times) := bits(number);
        number := number+1 ;
    end loop;
    return result;
end SHIFL_TO_R;

function IS_OVERFLOW( exp_bits: BIT_ARRAY;
                    precision: INTEGER)
return BOOLEAN is
variable result: BOOLEAN ;
begin
    case precision is
        when 32 => ----single precision
            if exp_bits =B"11111111" then
                result := TRUE;
            else
                result := FALSE;
            end if;
        when others => ----double precision
            if exp_bits =B"1111111111" then
                result := TRUE;
            else
                result := FALSE;
            end if;
    end case;
end IS_OVERFLOW;

```

```

        end if;
    end case;
    return result;
end IS_OVERFLOW;

function IS_UNDERFLOW( exp_bits: BIT_ARRAY;
                       precision: INTEGER)
    return BOOLEAN is
    variable result: BOOLEAN ;
    begin
        case precision is
            when 32 => -----single precision
                if exp_bits =B"00000000" then
                    result := TRUE;
                else
                    result := FALSE;
                end if;
            when others => ----double precision
                if exp_bits =B"000000000000" then
                    result := TRUE;
                else
                    result := FALSE;
                end if;
            end case;
        return result;
    end IS_UNDERFLOW;

function IS_ZERO( bits: BIT_ARRAY)
    return BOOLEAN is
    variable result: BOOLEAN ;
    begin
        for i in bits'range loop
            if bits(i) /= '0' then
                result := FALSE;
                return result ;
            end if;
        end loop ;
        result := TRUE ;
        return result;
    end IS_ZERO;

function IS_NAN( exp_bits: BIT_ARRAY )
    return BOOLEAN is
    variable result: BOOLEAN ;
    begin
        for i in exp_bits'range loop
            if exp_bits(i) /= '1' then
                result := FALSE;
                return result ;
            end if;
        end loop ;
        result := TRUE ;
        return result;
    end IS_NAN;

```

```

        end if ;
    end loop ;
    result := TRUE ;
    return result;
end IS_NAN ;

function BECOME_ZERO( bits: BIT_ARRAY)
return BIT_ARRAY is
variable result: BIT_ARRAY(bits'left downto bits'right);
begin
    for i in bits'range loop
        result(i) := '0';
    end loop ;
    return result;
end BECOME_ZERO;

function BECOME_NAN( bits: BIT_ARRAY)
return BIT_ARRAY is
variable result: BIT_ARRAY(bits'left downto bits'right);
begin
    for i in bits'range loop
        result(i) := '1';
    end loop ;
    return result;
end BECOME_NAN;

function SET_FLAG( bits,exp_bits: BIT_ARRAY ;
                    precision: INTEGER)
return FLAG is
variable result: FLAG ;
begin
    result.ovf_bit := '0';
    result.nan_bit := '0';
    result.zero_bit := '0';
    result.unf_bit := '0';
    if IS_OVERFLOW( exp_bits, precision) then
        result.ovf_bit := '1';
        result.nan_bit := '1';
    elsif IS_UNDERFLOW( exp_bits, precision) then
        result.unf_bit := '1';
        if IS_ZERO( bits) then
            result.zero_bit := '1';
        end if;
    end if;
    return result ;
end SET_FLAG;

function ADD(sign_a:BIT; bits_a: BIT_ARRAY; sign_b:BIT;

```

```

        bits_b : BIT_ARRAY)
return REAL is
variable result: REAL;
variable fra_a: REAL;
variable fra_b: REAL;
variable sig_a: REAL;
variable sig_b: REAL;
variable xbuff: BIT_ARRAY( 0 to 1);
begin
xbuff := sign_a&sign_b;
case xbuff is
when "00" =>
    sig_a := 1.0;
    sig_b := 1.0;
when "01" =>
    sig_a := 1.0;
    sig_b := -1.0;
when "10" =>
    sig_a := -1.0;
    sig_b := 1.0;
when "11" =>
    sig_a := -1.0;
    sig_b := -1.0;
end case;
fra_a := BITSARRAY_TO_FP(bits_a);
fra_b := BITSARRAY_TO_FP(bits_b);
result := abs(sig_a*fra_a + sig_b*fra_b) ;
return result;
end ADD;

```

```

function INCREASEMENT(bits:BIT_ARRAY; precision:INTEGER)
return BIT_ARRAY is
variable result : BIT_ARRAY( bits'length-1 downto 0 );
variable length : INTEGER := bits'length ;
variable buf : BIT_ARRAY( 0 to 1 );
variable carry : BIT := '1'; -- initial condition C(0)=1
variable bit_num :integer := 0;
begin
    if IS_OVERFLOW( bits,precision ) then
        result := bits ;
        return result;
    end if;
    while bit_num <= length-1 loop
        buf := bits(bit_num) & carry ;
        case buf is
        when "00" =>
            carry := '0';
            result(bit_num) :='0';
        when "01" =>
            carry := '0';

```

```

        result(bit_num) := '1';
    when "10" =>
        carry := '0';
        result(bit_num) := '1';
    when "11" =>
        carry := '1';
        result(bit_num) := '0';
    end case;
    bit_num := bit_num + 1;
end loop;
return result;
end INCREASEMENT ;

```

```

function DECREASEMENT(bits:BIT_ARRAY; precision:INTEGER)
return BIT_ARRAY is
variable result : BIT_ARRAY( bits'length-1 downto 0 );
variable length : INTEGER := bits'length ;
variable buf : BIT_ARRAY( 0 to 1 );
variable borrow:BIT := '1'; --initial condition C(0) = 1
variable bit_num :integer := 0;
begin
    if IS_UNDERFLOW( bits,precision ) then
        result := bits ;
        return result;
    end if;
    while bit_num <= length-1 loop
        buf := bits(bit_num) & borrow ;
        case buf is
            when "00" =>
                borrow := '0';
                result(bit_num) := '0';
            when "01" =>
                borrow := '1';
                result(bit_num) := '1';
            when "10" =>
                borrow := '0';
                result(bit_num) := '1';
            when "11" =>
                borrow := '0';
                result(bit_num) := '0';
        end case;
        bit_num := bit_num + 1;
    end loop;
    return result;
end DECREASEMENT ;

```

```

function BACK_TO_BITSARRAY(exp_bits:BIT_ARRAY;
fp:REAL; precision:INTEGER)
return BIT_ARRAY is

```

```

variable length:INTEGER := precision-1;
variable result: BIT_ARRAY(length-1 downto 0) ;
variable bits_buf: BIT_ARRAY(length-1-exp_bits'length
                               downto 0) ;

variable fra_value: REAL;
variable fp_buf :REAL := fp;
variable exp_bits_buf :BIT_ARRAY( exp_bits'length-1
                                   downto 0) := exp_bits;
---be careful input prarmeter must be positive real value --
begin
  if fp = 0.0 then
    result := BECOME_ZERO( result );
    return result;
  end if ;
  if( fp>1.0 and IS_OVERFLOW( exp_bits , precision)) then
    result := BECOME_NAN( result ) ;
    return result ;
  end if ;
  if ( fp<1.0 and IS_UNDERFLOW( exp_bits,precision)) then
    result := BECOME_ZERO( result );
    return result;
  else
    while abs( fp_buf-1.5 ) > 0.5 loop
      if fp_buf > 2.0 then
        fp_buf := fp_buf / 2.0;
        exp_bits_buf
          := INCREASEMENT( exp_bits_buf,precision);
        if IS_OVERFLOW( exp_bits_buf,precision) then
          exit when( fp_buf <= 2.0 and fp_buf >= 1.0);
          bits_buf := BECOME_ZERO( bits_buf);
          --set the fra_bits
          result := exp_bits_buf & bits_buf;
          -- become 0.
          return result;
        end if;

        elsif fp_buf < 1.0 then
          fp_buf := fp_buf * 2.0;
          exp_bits_buf :=
            DECREASEMENT( exp_bits_buf,precision);
          ----- if underflow condition occurred
          if IS_UNDERFLOW( exp_bits_buf,precision) then
            bits_buf := FP_TO_BITSARRAY(
              fp_buf,bits_buf'length );
            result := exp_bits_buf & bits_buf ;
            return result;
          end if ;
        end if;
      end loop; -- it produces value over between 1 and 2
      fra_value := fp_buf - 1.0;
      if fra_value = 1.0 then

```



```

        if IS_OVERFLOW( exp_bits_buf,precision) then
            bits_buf := BECOME_ZERO( bits_buf);
        else
            exp_bits_buf :=
                INCREASEMENT( exp_bits_buf,precision);
            bits_buf := BECOME_ZERO( bits_buf);
        end if;
    elsif fra_value = 0.0 then
        bits_buf := BECOME_ZERO( bits_buf);
    else
        bits_buf :=
            FP_TO_BITSARRAY( fra_value,bits_buf'length );
    end if;
    result := exp_bits_buf & bits_buf ;
end if;
return result;
end BACK_TO_BITSARRAY;

end refer ;

```

## APPENDIX B: THE TOP FUNCTIONS AND BEHAVIOR OF THE FPU

### A. THE TOP FUNCTIONS OF THE FPU

----- Floating Point Addition -----

```
library fpu;
use fpu.refer.all;
package FP_ADDER is

    function ADDER(sign_a:BIT; bits_a: BIT_ARRAY; sign_b:BIT;
                    bits_b : BIT_ARRAY ; exp_diff: INTEGER)
        return REAL;

    function ADD2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
                    exp_length,mantissa_length,precision: INTEGER )
        return BIT_ARRAY ;

end FP_ADDER ;

package body FP_ADDER is

    function ADDER(sign_a:BIT; bits_a: BIT_ARRAY; sign_b:BIT;
                    bits_b : BIT_ARRAY ; exp_diff :INTEGER)
        return REAL is
        variable result: REAL;
        variable fra_a: REAL;
        variable fra_b: REAL;
        variable sig_a: REAL;
        variable sig_b: REAL;
        variable xbuff: BIT_ARRAY( 0 to 1);
        begin
            xbuff := sign_a&sign_b;
            case xbuff is
            when "00" =>
                sig_a := 1.0;
                sig_b := 1.0;
            when "01" =>
                sig_a := 1.0;
                sig_b := -1.0;
            when "10" =>
                sig_a := -1.0;
                sig_b := 1.0;
            when "11" =>
                sig_a := -1.0;
```

```

    sig_b := -1.0;
end case;
if exp_diff >= 0 then
    fra_a := BITSARRAY_TO_FP(bits_a);
    fra_b := BITSARRAY_TO_FP(SHIFL_TO_R(bits_b, exp_diff));
else
    fra_a := BITSARRAY_TO_FP
        (SHIFL_TO_R(bits_a, abs(exp_diff)));
    fra_b := BITSARRAY_TO_FP(bits_b);
end if ;
result := abs(sig_a*fra_a + sig_b*fra_b) ;
return result;
end ADDER;

function ADD2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
exp_length, mantissa_length, precision: INTEGER )
    return BIT_ARRAY is
        variable a_is_nan : BOOLEAN;
        variable b_is_nan : BOOLEAN;
        variable a_is_zero : BOOLEAN;
        variable b_is_zero : BOOLEAN;
        variable a_is_underflow : BOOLEAN;
        variable b_is_underflow : BOOLEAN;
        variable exp_a : INTEGER;
        variable exp_b : INTEGER;
        variable exp_diff : INTEGER ;
        variable bits_length : INTEGER := bits_a'length;
        variable sign_bit_a : BIT := bits_a(bits_a'left);
        variable exp_bits_a : BIT_ARRAY(bits_a'left-1 downto
            bits_a'left-exp_length);
        variable mantissa_a : BIT_ARRAY(mantissa_length downto
            bits_a'right);
        variable sign_bit_b : BIT := bits_b(bits_b'left);
        variable exp_bits_b : BIT_ARRAY(bits_b'left-1 downto
            bits_b'left-exp_length);
        variable mantissa_b : BIT_ARRAY(mantissa_length downto
            bits_b'right);
        variable bits_c: BIT_ARRAY(bits_a'left downto
            bits_a'right);
        variable sign_bit_c : BIT ;
        variable exp_bits_c: BIT_ARRAY(bits_a'left-1 downto
            bits_a'left-exp_length);
        variable buf_bits_c : BIT_ARRAY( bits_a'left-1 downto
            bits_a'right);
        variable fra_c : REAL ;

begin
    exp_bits_a := bits_a(bits_a'left-1 downto
        bits_a'left-exp_length);
    exp_bits_b := bits_b(bits_b'left-1 downto
        bits_b'left-exp_length);

```

```

a_is_nan := IS_OVERFLOW( exp_bits_a, precision) ;
b_is_nan := IS_OVERFLOW( exp_bits_b, precision) ;
a_is_underflow := IS_UNDERFLOW( exp_bits_a, precision) ;
b_is_underflow := IS_UNDERFLOW( exp_bits_b, precision) ;
a_is_zero := IS_ZERO( bits_a );
b_is_zero := IS_ZERO( bits_b );
if a_is_zero then
    bits_c := bits_b;
    return bits_c;
elsif b_is_zero then
    bits_c := bits_a;
    return bits_c ;
end if ;
case ( a_is_nan or b_is_nan ) is
when TRUE =>
    if ( a_is_nan and a_is_nan ) then
        bits_c := bits_a;
    elsif b_is_nan then
        bits_c := bits_b;
    else
        bits_c := bits_a;
    end if;
when FALSE =>
    exp_a := BITSARRAY_TO_INT(exp_bits_a);
    exp_b := BITSARRAY_TO_INT(exp_bits_b);
    exp_diff := exp_a - exp_b ;
    if exp_diff >= 24 then
        bits_c := bits_a;
        return bits_c ;
    elsif abs(exp_diff) >= 24 then
        bits_c := bits_b;
        return bits_c;
    end if ;
    if exp_diff > 0 then
        exp_bits_c := exp_bits_a ;
        sign_bit_c := sign_bit_a ;
    elsif( exp_diff < 0 ) then
        exp_bits_c := exp_bits_b ;
        sign_bit_c := sign_bit_b ;
    end if;
    if ( a_is_underflow or b_is_underflow ) then
        if a_is_underflow then
            ---in the underformat there is not unhidden bit exitent

            mantissa_a := '0' & bits_a( mantissa_length-1 downto
                                         bits_a'right);

            elsif b_is_underflow then
                mantissa_b := '0' & bits_b( mantissa_length-1
                                         downto bits_b'right);
            end if;

```

```

        else
mantissa_a :=UNHIDDEN_BIT(bits_a( mantissa_length-1
                                downto bits_a'right));
mantissa_b :=UNHIDDEN_BIT(bits_b( mantissa_length-1
                                downto bits_b'right));
        end if ;

if( exp_diff = 0 and ( mantissa_a >= mantissa_b )) then
    exp_bits_c := exp_bits_a ;
    sign_bit_c := sign_bit_a ;
elsif( exp_diff = 0 and ( mantissa_b > mantissa_a ))
then
    exp_bits_c := exp_bits_b ;
    sign_bit_c := sign_bit_b ;
end if ;

fra_c :=2.0 * ADDER( sign_bit_a, mantissa_a,
                    sign_bit_b, mantissa_b,exp_diff);
if fra_c = 0.0 then
    bits_c := BECOME_ZERO( bits_a );
else
    buf_bits_c := BACK_TO_BITSARRAY( exp_bits_c,
                                    fra_c,precision );
    bits_c := sign_bit_c & buf_bits_c ;
end if ;
end case;
return bits_c ;
end ADD2;

end FP_ADDER ;

```

----- Floating Point Subtraction -----

```

library fpu;
use fpu.refer.all;
use fpu.fp_adder.all;
package FP_SUBER is

    function SUB2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
                    exp_length, mantissa_length, precision: INTEGER )
        return BIT_ARRAY ;

end FP_SUBER ;

```

```

package body FP_SUBER is

function SUB2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
               exp_length, mantissa_length, precision: INTEGER )
               return BIT_ARRAY is

variable buf_bits_b : BIT_ARRAY(bits_b'left downto
                                bits_b'right)
                                := bits_b ;
variable bits_c : BIT_ARRAY(bits_b'left downto
                             bits_b'right);
begin
  if bits_b(bits_b'left) = '1' then
    buf_bits_b( bits_b'left) := '0';
  else
    buf_bits_b( bits_b'left) := '1';
  end if;
  bits_c := ADD2(bits_a, buf_bits_b, exp_length,
                 mantissa_length , precision );
  return bits_c ;
end SUB2 ;
end FP_SUBER ;

```

----- Floating Point Multiplication -----

```

library fpu;
use fpu.refer.all;
package FP_MULTIER is

function MULTI2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
                 exp_length,mantissa_length,precision: INTEGER )
                 return BIT_ARRAY ;

end FP_MULTIER ;

package body FP_MULTIER is

```

```

function MULTI2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
                 exp_length,mantissa_length,precision: INTEGER )
                 return BIT_ARRAY is
  variable a_is_zero :BOOLEAN;
  variable b_is_zero :BOOLEAN;
  variable a_is_nan :BOOLEAN;
  variable b_is_nan :BOOLEAN;
  variable a_is_underflow :BOOLEAN;
  variable b_is_underflow :BOOLEAN;
  variable exp_a :INTEGER,
  variable exp_b :INTEGER;

```

```

variable exp_sum :INTEGER ;
variable bits_length :INTEGER := bits_a'length;
variable sign_bit_a : BIT := bits_a(bits_a'left);
variable exp_bits_a : BIT_ARRAY(bits_a'left-1 downto
                                bits_a'left-exp_length);
variable mantissa_a : BIT_ARRAY(mantissa_length downto
                                bits_a'right);
variable sign_bit_b : BIT := bits_b(bits_b'left);
variable exp_bits_b : BIT_ARRAY(bits_b'left-1 downto
                                bits_b'left-exp_length);
variable mantissa_b : BIT_ARRAY(mantissa_length downto
                                bits_b'right);
variable bits_c: BIT_ARRAY(bits_a'left downto
                            bits_a'right);
variable sign_bit_c : BIT ;
variable exp_bits_c:BIT_ARRAY(bits_a'left-1 downto
                              bits_a'left-exp_length);
variable buf_bits_c :BIT_ARRAY( bits_a'left-1 downto
                              bits_a'right);
variable fra_c : REAL ;

begin
sign_bit_c := sign_bit_a xor sign_bit_b ;
exp_bits_a := bits_a(bits_a'left-1 downto
                    bits_a'left-exp_length);

exp_bits_b := bits_b(bits_b'left-1 downto
                    bits_b'left-exp_length);

a_is_zero := IS_ZERO( exp_bits_a );
b_is_zero := IS_ZERO( exp_bits_b );
if (a_is_zero or b_is_zero) then
    bits_c := BECOME_ZERO( bits_c );
    bits_c( bits_c'length-1 ):= sign_bit_c ;
else
    a_is_nan := IS_OVERFLOW( exp_bits_a, precision) ;
    b_is_nan := IS_OVERFLOW( exp_bits_b, precision) ;
    a_is_underflow:= IS_UNDERFLOW( exp_bits_a,precision);
    b_is_underflow:= IS_UNDERFLOW( exp_bits_b,precision);
    case ( a_is_nan or b_is_nan ) is
        when TRUE =>
            if a_is_nan then
                bits_c := BECOME_NAN( bits_a );
                bits_c( bits_c'length-1 ):= sign_bit_c ;
            else
                bits_c := BECOME_NAN( bits_b );
                bits_c( bits_c'length-1 ):= sign_bit_c ;
            end if;
        when FALSE =>
            exp_a := BITSARRAY_TO_INT(exp_bits_a);

```

```

exp_b := BITSARRAY_TO_INT(exp_bits_b);

if( a_is_underflow or b_is_underflow )then
  if a_is_underflow then
-- in underflow formate there is not unhidden bit existing
mantissa_a := '0' & bits_a(mantissa_length-1 downto
                           bits_a'right);
    elsif b_is_underflow then
      mantissa_b := '0' & bits_b( mantissa_length-1 downto
                                   bits_b'right);
    end if;
  else
    mantissa_a := UNHIDDEN_BIT(bits_a( mantissa_length-1
                                         downto bits_a'right));
    mantissa_b := UNHIDDEN_BIT(bits_b( mantissa_length-1
                                         downto bits_b'right));
  end if;
fra_c := 4.0 * BITSARRAY_TO_FP( mantissa_a ) *
          BITSARRAY_TO_FP( mantissa_b );

exp_sum := exp_a + exp_b ;

if precision = 32 then      -----single precision
  exp_sum := exp_sum - 127; ----IEEE EXP FORMAT
  if exp_sum >= 255 then
    bits_c := BECOME_NAN( bits_c ) ;
    ----- overflow
    bits_c( bits_c'length-1 ):= sign_bit_c ;
  elsif exp_sum < 0 then
    if (exp_sum < -1) or ( exp_sum = -1 and
      bits_c < 2.0) then
      bits_c := BECOME_ZERO( bits_c ) ;
      ----underflow
      bits_c( bits_c'length-1 ):= sign_bit_c ;
      return bits_c ;
    elsif ( exp_sum = -1 and fra_c >= 2.0 )
    then
      fra_c := fra_c/2.0 ;
      exp_bits_c := B"00000000" ;
    end if ;
  else
    exp_bits_c := INT_TO_BITSARRAY( exp_sum
                                     ,exp_length) ;
  end if;
else

```



```

exp_sum := exp_sum - 1023;
---the other case is 64(double precision);

if exp_sum >= 2047 then
    bits_c := BECOME_NAN( bits_c ) ;
    ---- overflow

    bits_c( bits_c'length-1 ):= sign_bit_c ;
elsif exp_sum < 0 then

    if (exp_sum < -1) or ( exp_sum = -1 and
        fra_c < 2.0) then
        bits_c := BECOME_ZERO( bits_c ) ;
        ---underflow

        bits_c( bits_c'length-1 ):= sign_bit_c ;
        return bits_c ;
    elsif ( exp_sum = -1 and fra_c >= 2.0 )
    then
        fra_c := fra_c/2.0 ;
        exp_bits_c := B"000000000000" ;
    end if ;

else
    exp_bits_c := INT_TO_BITSARRAY( exp_sum
                                    ,exp_length) ;

    end if;
end if ;
buf_bits_c := BACK_TO_BITSARRAY( exp_bits_c,
                                fra_c, precision );

bits_c := sign_bit_c & buf_bits_c ;

end case;
end if;
return bits_c ;
end MULTI2;
end FP_MULTIER ;

```

----- Floating Point Divider -----

```

library fpu;
use fpu.refer.all;
package FP_DIVIDER is

    function DIVIDE2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
        exp_length,mantissa_length,precision: INTEGER )
        return BIT_ARRAY ;

```

```

function DIV( bits_a, bits_b : BIT_ARRAY ; exp_length
               , precision : INTEGER )
               return BIT_ARRAY ;

end FP_DIVIDER ;

package body FP_DIVIDER is

function DIV( bits_a, bits_b : BIT_ARRAY ; exp_length
               , precision : INTEGER)
               return BIT_ARRAY is
    variable length : INTEGER := bits_a'length ;
    variable diff_exp_value : INTEGER ;
    variable exp_bits_a_value : INTEGER ;
    variable exp_bits_b_value : INTEGER ;
    variable fra_bits_b_value : REAL ;
    variable fra_bits_a_value : REAL ;
    variable fra_bits_c_value : REAL ;
    variable bits_value : REAL ;
    variable sign_bits_a : BIT := bits_a( bits_a'left ) ;
    variable sign_bits_b : BIT := bits_b( bits_b'left ) ;
    variable sign_bits_c : BIT ;
    variable bits_c : BIT_ARRAY( bits_a'left downto
                                   bits_a'right ) ;
    variable buf_bits_c : BIT_ARRAY( bits_a'left -1 downto
                                       bits_a'right ) ;
    variable exp_bits_b : BIT_ARRAY( bits_b'left-1 downto
                                       bits_b'left-exp_length )
                                   := bits_b( bits_b'left-1 downto
                                               bits_b'left-exp_length ) ;
    variable exp_bits_a : BIT_ARRAY( bits_a'left-1 downto
                                       bits_a'left-exp_length )
                                   := bits_a( bits_a'left-1 downto
                                               bits_a'left-exp_length ) ;

    variable exp_bits_buf : BIT_ARRAY( bits_a'left-1
                                         downto bits_a'left-exp_length ) ;

begin
    sign_bits_c := sign_bits_a xor sign_bits_b ;
    exp_bits_b_value := BITSARRAY_TO_INT( exp_bits_b ) ;

    exp_bits_a_value := BITSARRAY_TO_INT( exp_bits_a ) ;

    if ( IS_UNDERFLOW( exp_bits_a, precision ) )
       or ( IS_OVERFLOW( exp_bits_b, precision ) ) then

        buf_bits_c := BECOME_ZERO( buf_bits_c ) ;
        bits_c := sign_bits_c & buf_bits_c ;
        return bits_c ;
    end if ;
end DIV ;

```

```

elseif ( IS_OVERFLOW( exp_bits_a,precision ))
  or ( IS_UNDERFLOW( exp_bits_b,precision )) then

  buf_bits_c := BECOME_NAN( buf_bits_c ) ;
  bits_c := sign_bits_c & buf_bits_c ;
  return bits_c ;

else
  fra_bits_a_value :=BITSARRAY_TO_FP(
    UNHIDDEN_BIT(bits_a( bits_a'left
      - exp_length-1 downto bits_a'right ))) ;

  fra_bits_b_value :=BITSARRAY_TO_FP(
    UNHIDDEN_BIT(bits_b( bits_b'left
      - exp_length-1 downto bits_b'right ))) ;

end if;

fra_bits_c_value := fra_bits_a_value /
  fra_bits_b_value ;

if precision = 32 then          --single precision
  diff_exp_value := exp_bits_a_value -
    exp_bits_b_value + 127;

  if (diff_exp_value > 255 or
    (diff_exp_value = 255 and
    fra_bits_c_value >= 1.0)) then
    buf_bits_c := BECOME_NAN( buf_bits_c ) ;
    bits_c := sign_bits_c & buf_bits_c ;
    return bits_c ;
  elseif( diff_exp_value < 0 or
    ( diff_exp_value = 0 and
    fra_bits_c_value <= 1.0)) then
    buf_bits_c := BECOME_ZERO( buf_bits_c );
    bits_c := sign_bits_c & buf_bits_c ;
    return bits_c ;
  else
    exp_bits_buf:= INT_TO_BITSARRAY(
      diff_exp_value, exp_length);
  end if;

else
  diff_exp_value := exp_bits_a_value -
    exp_bits_b_value + 1023;
    ----double precision

  if (diff_exp_value > 2047 or

```

```

        (diff_exp_value = 2047 and fra_bits_c_value
          >= 1.0)) then
          buf_bits_c := BECOME_NAN( buf_bits_c ) ;
          bits_c := sign_bits_c & buf_bits_c ;
          return bits_c ;
        elsif( diff_exp_value < 0 or
          ( diff_exp_value = 0 and
            fra_bits_c_value <= 1.0)) then
          buf_bits_c := BECOME_ZERO( buf_bits_c );
          bits_c := sign_bits_c & buf_bits_c ;
          return bits_c ;
        else
          exp_bits_buf:= INT_TO_BITSARRAY(
            diff_exp_value, exp_length);
        end if;

    end if ;

    buf_bits_c := BACK_TO_BITSARRAY(
      exp_bits_buf, fra_bits_c_value,precision );
    bits_c := sign_bits_c & buf_bits_c ;
    return bits_c;
  end DIV ;

```

```

function DIVIDE2( bits_a: BIT_ARRAY ; bits_b: BIT_ARRAY;
  exp_length,mantissa_length,precision: INTEGER )
  return BIT_ARRAY is
    variable a_is_zero :BOOLEAN;
    variable b_is_zero :BOOLEAN;
    variable a_is_nan :BOOLEAN;
    variable b_is_nan :BOOLEAN;
    variable inv_bits_b: BIT_ARRAY(bits_b'left downto
      bits_b'right);
    variable bits_c: BIT_ARRAY(bits_a'left downto
      bits_a'right);
    variable sign_bit_c : BIT ;
    variable exp_bits_a:BIT_ARRAY(bits_a'left-1 downto
      bits_a'left-exp_length)
      :=bits_a(bits_a'left-1 downto
        bits_a'left-exp_length);
    variable exp_bits_b:BIT_ARRAY(bits_b'left-1 downto
      bits_b'left-exp_length)
      :=bits_b(bits_b'left-1 downto
        bits_b'left-exp_length);

  begin
    a_is_zero := IS_ZERO( exp_bits_a );
    b_is_zero := IS_ZERO( exp_bits_b );

```

```

if a_is_zero then
    bits_c := BECOME_ZERO( bits_a );
elsif ( not( a_is_zero) and b_is_zero ) then
    bits_c := BECOME_NAN( bits_a );
else
    a_is_nan := IS_OVERFLOW( exp_bits_a, precision) ;
    b_is_nan := IS_OVERFLOW( exp_bits_b, precision) ;
    case ( a_is_nan or b_is_nan ) is
        when TRUE =>
            if b_is_nan then
                bits_c := BECOME_ZERO( bits_a );
            else
                bits_c := bits_a ;
            end if;
        when FALSE =>
            bits_c := DIV( bits_a, bits_b, exp_length,
                           precision);
    end case;
end if;
return bits_c ;
end DIVIDE2;
end FP_DIVIDER ;

```

## B. THE BEHAVIOR FUNCTIONS OF THE FPU

```

library fpu;
use fpu.refer.all,    fpu.fp_adder.all,    fpu.fp_suber.all,
   fpu.fp_multier.all,
   fpu.fp_divider.all;
package utility1 is

    function FP_UNIT( bits_a, bits_b: BIT_ARRAY;
        precision, choice : INTEGER ) return BIT_ARRAY ;

end utility1 ;

package body utility1 is

    function FP_UNIT( bits_a, bits_b: BIT_ARRAY;
        precision, choice : INTEGER) return BIT_ARRAY is

        variable exp_length : INTEGER ;
        variable mantissa_length : INTEGER ;
        variable buf_c : BIT_ARRAY( bits_a'left downto
                                       bits_a'right );

    begin
        if precision = 32 then
            exp_length := 8;

```

```

        mantissa_length := 23 ;
    else
        exp_length := 11;      ----double precision
        mantissa_length := 52;
    end if;
    case choice is
    when 1 =>
        buf_c := ADD2( bits_a , bits_b , exp_length,
                        mantissa_length, precision);
    when 2 =>
        buf_c := SUB2( bits_a , bits_b , exp_length,
                        mantissa_length, precision);
    when 3 =>
        buf_c := MULTI2( bits_a , bits_b , exp_length,
                          mantissa_length, precision);
    when others =>
        buf_c := DIVIDE2( bits_a , bits_b , exp_length,
                           mantissa_length, precision);
    end case ;
    return buf_c;
end FP_UNIT;

end utility1 ;

```

# APPENDIX C: THE SOURCE FILE OF THE FPU CHIP AMD29325

```
library fpu;
use fpu.refer.all, fpu.utility1.all;
----- it is designed with single precision and
only 4
----- arithmetic operations built in AMD29325
```

```
entity AM29325 is
  generic( D_FPU_T : time := 110ns );
  port( R,S : in BIT_ARRAY( 31 downto 0)
        := B"00000000000000000000000000000000";
        ENR,ENS,ENY,ONEBUS,FT0,FT1,CLK : in BIT
        := '0';
        OE : in BOOLEAN := false ;
        IO_I2 : in BIT_ARRAY( 2 downto 0)
        := B"000" ;
        I3_I4 : in BIT_ARRAY( 1 downto 0)
        := B"00" ;
        IEEE_OR_DEC : in BIT
        := '1' ;
        S16_OR_S32, PROJ_OR_AFF : in BIT
        := '0' ;
        RND0_RND1: in BIT_ARRAY( 1 downto 0)
        := B"00" ;
        F : out BIT_ARRAY( 31 downto 0)
        := B"00000000000000000000000000000000" ;
        ovf, unf, zero, nan, invd, inet : out BIT
        := '0' ) ;
end AM29325 ;
```

```
library fpu;
use fpu.refer.all, fpu.utility1.all, fpu.write_file.all;
architecture behavioral of AM29325 is
```

```
begin
  process(CLK,OE)
    variable precision : INTEGER := R'length ;
    variable BUF_F : BIT_ARRAY( 31 downto 0) ;
    variable BUF_F_FLAG : FLAG ;
    variable choice : INTEGER ;
    constant ADD : INTEGER := 1;
    constant SUB : INTEGER := 2;
    constant MULTI : INTEGER := 3;
    constant DIV : INTEGER := 4;
```

```

begin
  if ( OE and (CLK'EVENT and CLK = '1' )) then
    case IO_I2 is
      when B"000" =>
        choice := ADD ;
      when B"001" =>
        choice := SUB ;
      when B"010" =>
        choice := MULTI ;
      when others =>
        choice := DIV ;
    end case ;
    BUF_F := FP_UNIT(R,S,precision,choice) ;
    F      <= BUF_F after D_FPU_T;
    BUF_F_FLAG := SET_FLAG(BUF_F, BUF_F(30 downto
      23),precision);
    ovf <= BUF_F_FLAG.ovf_bit after D_FPU_T ;
    unf <= BUF_F_FLAG.unf_bit after D_FPU_T ;
    zero<= BUF_F_FLAG.zero_bit after D_FPU_T ;
    nan <= BUF_F_FLAG.nan_bit after D_FPU_T ;
  end if ;
end process ;

end behavioral;

```



**THE APPENDIX D: THE SIMPLIFIED I/O PORT OF THE FPU CHIP  
AMD29325**

```

library fpu, fft ;
use fpu.refer.all, fft.AM29325 ;

    --- this program is created for simplifing
    --- AM29325 entity.

entity A29325 is

    generic ( D_FPU_T : TIME := 110 ns );
    port( in1,in2 : in BIT_ARRAY( 31 downto 0)
        -- in1, in2 input signal
        := B"00000000000000000000000000000000";
        clock : in BIT := '1' ;
        option : in INTEGER := 1 ;
        enable : in BOOLEAN := FALSE ;
        out1 : out BIT_ARRAY( 31 downto 0)
        := B"00000000000000000000000000000000" );
        -- output of fft

end A29325 ;

library fpu ,fft;
use fpu.refer.all, fft.am29325 ;

architecture simple of A29325 is

component AM29325
    generic( D_FPU_T : time := 110ns );
    port( R,S : in BIT_ARRAY( 31 downto 0)
        := B"00000000000000000000000000000000";
        ENR,ENS,ENY,ONEBUS,FT0,FT1,CLK : in BIT
        := '0';
        OE : in BOOLEAN := false ;
        I0_I2 : in BIT_ARRAY( 2 downto 0)
        := B"000" ;
        I3_I4 : in BIT_ARRAY( 1 downto 0)
        := B"00" ;
        IEEE_OR_DEC : in BIT
        := '1' ;
        S16_OR_S32, PROJ_OR_AFF : in BIT
        := '0' ;
        RND0_RND1: in BIT_ARRAY( 1 downto 0)
        := B"00" ;
    end port;
end component AM29325;

    -- Component instantiation and connections would follow here

```

```

        F : out BIT_ARRAY( 31 downto 0)
            := B"00000000000000000000000000000000" ;
        ovf, unf, zero, nan, invd, inet : out BIT
            := '0' ) ;
end component ;

for F1 : AM29325 use entity fft.AM29325( behavioral ) ;

signal  ENR,ENS,ENY,ONEBUS,FT0,FT1,CLK : BIT := '0';
signal  I3_I4 : BIT_ARRAY( 1 downto 0)      := B"00" ;
signal  IEEE_OR_DEC : BIT                    := '1' ;
signal  S16_OR_S32, PROJ_OR_AFF : BIT       := '0' ;
signal  RND0_RND1: BIT_ARRAY( 1 downto 0) := B"00" ;
signal  ovf, unf, zero, nan, invd, inet : BIT := '0' ;
signal  func : BIT_ARRAY( 2 DOWNT0 0) := "000" ;

begin

    process( option )
    begin
        if ( option = 1) then
            func <= "000" ;
        elsif( option = 2) then
            func <= "001" ;
        elsif( option = 3) then
            func <= "010" ;
        elsif( option = 4) then
            func <= "011" ;
        end if ;
    end process ;

    F1: AM29325
    generic map( D_FPU_T => 110ns )
    port map( in1, in2, ENR, ENS, ENY, ONEBUS, FT0, FT1,
              clock, enable, func, I3_I4, IEEE_OR_DEC,
              S16_or_S32, PROJ_OR_AFF, RND0_RND1, OUT1, ovf,
              unf, zero, nan, invd, inet ) ;

end simple ;

```

## APPENDIX E: THE PIPELINE STRUCTURE OF THE FFT BUTTERFLY

```

library fpu,fft;
use fpu.refer.all, fft.A29325, fft.basic.all ;

----- it designed for single precision

entity FFT_CELL is
  generic ( D_FPU_T : TIME := 110 ns );
  port( a_real,a_img : in LOGIC_ARRAY( 31 downto 0);
        -- a is the input signal.
        b_real,b_img : in LOGIC_ARRAY( 31 downto 0);
        -- b is the input signal.
        w_real,w_img : in LOGIC_ARRAY( 31 downto 0);
        -- w is the weight signal.
        clock : in BIT := '1' ;
        enable : in BOOLEAN := false ;
        -- chip enable for am29325
        ie : in BOOLEAN := FALSE ;

        -- input enable for final stage
        -- output
        oe : in BOOLEAN := FALSE ;
        -- output enable for first stage
        -- input
        c_real,c_img : out LOGIC_ARRAY( 31 downto 0) ;
        -- c is the output signal.

        d_real,d_img : out LOGIC_ARRAY( 31 downto 0));
        -- d is the
        -- output signal.

end FFT_CELL ;

library fpu, fft;
use fpu.refer.all, fft.A29325, fft.basic.all ;
architecture structural of FFT_CELL is

  component A29325
    generic ( D_FPU_T : TIME := 110 ns );
    port( in1,in2 : in BIT_ARRAY( 31 downto 0) -- in1, in2 is the
          input signal
          := B"00000000000000000000000000000000";
          clock : in BIT := '1' ;
          ----- rising edge trigger
          option : in INTEGER ;
          enable : in BOOLEAN := FALSE ;
  end component A29325;

```

```

-- chip enable for am29325
out1      : out BIT_ARRAY( 31 downto 0) );
-- output of fft

end component ;

for ALL : A29325 use entity fft.A29325( simple ) ;

signal buf_a_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal buf_b_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal buf_w_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal buf_a_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal buf_b_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal buf_w_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;

signal reg_1_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_1_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_2_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_2_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_3_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_3_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c1_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c1_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c2_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c2_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c3_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c3_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c4_real : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;
signal reg_c4_img  : BIT_ARRAY( 31 DOWNTO 0)
:= B"00000000000000000000000000000000" ;

```

```

signal reg_w1_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal reg_w1_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal reg_w2_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal reg_w2_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;

signal x1_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x1_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x2_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x2_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x3_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x3_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x4_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal x4_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal xc1_real : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;
signal xc1_img  : BIT_ARRAY( 31 DOWNT0 0)
                        := B"00000000000000000000000000000000" ;

signal div      : INTEGER := 4 ;    --- division
signal mult     : INTEGER := 3 ;    --- multiplication
signal sub      : INTEGER := 2 ;    --- subtraction
signal add      : INTEGER := 1 ;    --- addition

constant DEL_T1 : time      := 10 ns ;
constant DEL_T2 : time      := 110 ns ;

begin

----- begin at stage 1 -----
---- simply discribe D-FF behavior --

process( clock, ie )
begin
if ( clock'event and ( clock = '0' ) and ( ie = true)) then
    buf_a_real <= LOGIC_TO_BIT( a_real ) after DEL_T1;
    buf_a_img  <= LOGIC_TO_BIT( a_img  ) after DEL_T1;
    buf_b_real <= LOGIC_TO_BIT( b_real ) after DEL_T1;

```

```

        buf_b_img  <= LOGIC_TO_BIT( b_img  )  after DEL_T1;
        buf_w_real <= LOGIC_TO_BIT( w_real )  after DEL_T1;
        buf_w_img  <= LOGIC_TO_BIT( w_img  )  after DEL_T1;
    end if ;
end process;

----- end  of  stage 1 -----

----- begin  at  stage 2 -----

A1 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (    buf_a_real, buf_b_real, clock, sub,
enable, x1_real );

A2 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (    buf_a_img, buf_b_img, clock, sub,
enable, x1_img );

A3 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (    buf_a_real, buf_b_real, clock, add,
enable, xcl_real );

A4 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (    buf_a_img, buf_b_img, clock, add,
enable, xcl_img );

--- delay time at input weight factor
process( clock )
begin
    if ( clock'event and ( clock ='1' )) then
        reg_wl_real <= buf_w_real after DEL_T2;
        reg_wl_img  <= buf_w_img  after DEL_T2;
    end if ;
end process ;

----- end  of  stage 2 -----

----- begin  at  stage 3 -----
---- simply discribe D-FF behavior  --

```

```

process( clock )
begin
if ( clock'event and ( clock ='0' )) then
    reg_l_real  <= x1_real  after DEL_T1;
    reg_l_img   <= x1_img   after DEL_T1;
    reg_cl_real <= xcl_real after DEL_T1;
    reg_cl_img  <= xcl_img  after DEL_T1;
    reg_w2_real <= reg_w1_real after DEL_T1;
    reg_w2_img  <= reg_w1_img after DEL_T1;
end if ;
end process ;

----- end of stage 3 -----

----- begin at stage 4 -----

B1 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map ( reg_l_real, reg_w2_real, clock, mult,
        enable, x2_real );

B2 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map ( reg_l_img, reg_w2_real, clock, mult,
        enable, x2_img );

B3 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map ( reg_l_img, reg_w2_img, clock, mult,
        enable, x3_real );

B4 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map ( reg_l_real, reg_w2_img, clock, mult,
        enable, x3_img );

--- delay time at input weight factor
process( clock )
begin
if ( clock'event and ( clock ='1' )) then
    reg_c2_real <= reg_cl_real after DEL_T2;
    reg_c2_img  <= reg_cl_img  after DEL_T2;
end if ;
end process ;

----- end of stage 4 -----

```

```

----- begin at stage 5 -----
---- simply discribe D-FF behavior --
process( clock )
begin
  if ( clock'event and ( clock ='0' )) then
    reg_2_real  <= x2_real after DEL_T1;
    reg_2_img   <= x2_img  after DEL_T1;
    reg_3_real  <= x3_real after DEL_T1;
    reg_3_img   <= x3_img  after DEL_T1;
    reg_c3_real <= reg_c2_real after DEL_T1;
    reg_c3_img  <= reg_c2_img after DEL_T1;
  end if ;
end process ;
----- end of stage 5 -----

----- begin at stage 6 -----

C1 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (  reg_2_real, reg_3_real, clock, sub,
                enable, x4_real );

C2 : A29325
    generic map ( D_FPU_T =>110 ns)
    port map (  reg_2_img, reg_3_img, clock, add,
                enable, x4_img );

--- delay time at input weight factor
process( clock )
begin
  if ( clock'event and ( clock ='1' )) then
    reg_c4_real <= reg_c3_real after DEL_T2;
    reg_c4_img  <= reg_c3_img  after DEL_T2;
  end if ;
end process ;

----- end of stage 6 -----

```



```

----- begin at stage 7 -----
---- simply discribe D-FF behavior --
process( clock, oe )
begin
  if ( clock'event and ( clock ='0' ) and ( oe = true ) )
  then
    c_real <= BIT_TO_LOGIC( reg_c4_real ) after DEL_T1;
    c_img  <= BIT_TO_LOGIC( reg_c4_img  ) after DEL_T1;
    d_real <= BIT_TO_LOGIC( x4_real    ) after DEL_T1;
    d_img  <= BIT_TO_LOGIC( x4_img     ) after DEL_T1;
  end if ;
end process ;

----- end of stage 7 -----

end structural;

```

## APPENDIX F: THE ADDRESS SEQUENCE GENERATOR AND CONTROLLER

```
library fpu, fft;
use      fpu.refer.all,    fft.basic.all,    fft.ram_256,
fft.convert.all ;
entity SEQ_CONT is
  generic( test_number : positive := 2 ) ;--- from 1 to 6 ---
end ;
```

```
library fpu, fft;
use      fpu.refer.all,    fft.basic.all,    fft.ram_256,
fft.convert.all ;
architecture simple of SEQ_CONT is
```

```
function RESOLVE( bits_1, bits_2: LOGIC_ARRAY)
  return LOGIC_ARRAY is
  variable result :LOGIC_ARRAY( bits_1'left downto
                                bits_1'right) ;
  variable test1 : BOOLEAN ;
  variable test2 : BOOLEAN ;
begin
  test1 := IS_HI_Z_OR_X( bits_1 ) ;
  test2 := IS_HI_Z_OR_X( bits_2 ) ;
  if( test1 and test2 ) then
    for i in bits_1'range loop
      result(i):= 'X' ;
    end loop ;
  elsif( test1 ) then
    result := bits_2 ;
  elsif( test2 ) then
    result := bits_1 ;
  else
    assert( test1 and test2 )
    report " bus can not resolve any one input signal "
    severity error ;
  end if ;
  return result ;
end RESOLVE ;
```

```
function TABLE1( bits: BIT_ARRAY) return INTEGER is
  variable result :integer := 0 ;
begin
  result := 2** ( BITSARRAY_TO_INT( bits)+ 1) ;
  return result ;

end TABLE1 ;
```

```

function TABLE2( N: INTEGER) return INTEGER is
variable result :integer := 0 ;
begin

    while 2**( result) < N loop
        result  := result + 1 ;
    end loop ;
    return result ;

end TABLE2 ;

```

```

constant    chs_setup_t : TIME := 200 ns ;
constant    wrt_setup_t : TIME := 200 ns ;

signal      LEN          : BIT_ARRAY( 2 DOWNT0 0 ) := "000" ;

signal      ISTO         : BIT := '1'      ;
signal      CHE          : BIT := '1'      ;
signal      IN_R         : BIT := '0'      ;
signal      OUT_A        : BIT := '0'      ;

signal      IN_E         : BIT := '1'      ;
signal      OUT_E        : BIT := '1'      ;
signal      FFT_CMP      : BIT := '0'      ;
signal      STAGE_CNT    : INTEGER := -1 ;
signal      OSTO         : BIT := '1'      ;
signal      TRIG         : BIT := '0'      ;
signal      EN           : BIT := '1'      ;
signal      S0           : BIT := '0'      ;
signal      S1           : BIT := '0'      ;
signal      ADDR_0       : LOGIC_ARRAY( 7 downto 0)
                        := "ZZZZZZZZ";
signal      CHS_0        : BIT := '1'      ;
signal      RW_0         : BIT := '1'      ;
signal      ADDR_WC      : LOGIC_ARRAY( 7 downto 0)
                        := "ZZZZZZZZ";
signal      CHS_WC       : BIT := '1'      ;
signal      RW_WC        : BIT := '1'      ;
signal      ADDR_1       : LOGIC_ARRAY( 7 downto 0)
                        := "ZZZZZZZZ";
signal      CHS_1        : BIT := '1'      ;
signal      RW_1         : BIT := '1'      ;

signal      TRIG_RD_0    : BIT      := '0' ;
signal      TRIG_WR_0    : BIT      := '0' ;

```

```

signal    TRIG_RD_1  : BIT      := '0' ;
signal    TRIG_WR_1  : BIT      := '0' ;
signal    RD_ADDR_0  : LOGIC_ARRAY( 7 DOWNT0 0)
           := "ZZZZZZZZ" ;
signal    RD_ADDR_1  : LOGIC_ARRAY( 7 DOWNT0 0)
           := "ZZZZZZZZ" ;
signal    WR_ADDR_0  : LOGIC_ARRAY( 7 DOWNT0 0)
           := "ZZZZZZZZ" ;
signal    WR_ADDR_1  : LOGIC_ARRAY( 7 DOWNT0 0)
           := "ZZZZZZZZ" ;

signal    IE          : BOOLEAN := FALSE ;
signal    OE          : BOOLEAN := FALSE ;
signal    ENABLE       : BOOLEAN := FALSE ;
signal    STATE       : INTEGER := 0 ;

```

```
begin
```

```

-----
----- FFT controller -----
process( CLOCK, IN_E, OUT_E )
variable CNT : INTEGER := 0 ;
begin
    if ( (IN_E='0' and IN_E'event ) and
          ( OUT_E='0'and OUT_E'event ) ) then
        CNT := 0 ;
        IN_R <= '1' ;
        OUT_A <= '0' ;
        IE <= TRUE ;
        ENABLE <= TRUE ;
    elsif(( CLOCK'event and CLOCK = '0')) then
        CNT := CNT + 1 ;
        if( CNT = 4 ) then
            OE <= TRUE ;
        elsif( CNT = 5 ) then
            OUT_A <= '1' ;
        end if ;
    elsif( (CNT >=4) and (OUT_E = '1') and (CLOCK'event))
    then
        OUT_A <= '0' ;
        ENABLE <= FALSE ;
        OE <= FALSE after 500 ns ;
    elsif( (CNT >=4) and ( IN_E ='1') ) then
        IN_R <= '0' ;
        IE <= FALSE ;
    end if ;
end process ;

```

```

-----
--
-----address sequencer -----
----- generate step by step signal -----
process( CLOCK, LEN, CHE, STATE, IN_R, OUT_A )
variable R_CNT      : INTEGER := 0 ;
variable W_CNT      : INTEGER := 0 ;
variable N          : INTEGER := 0 ;
variable PTR        : BIT      := '0' ;
variable COE_BUF    : LOGIC_ARRAY( 7 downto 0 )
                    := "00000000" ;
variable F          : INTEGER := 0 ;
begin
    if ( ( CHE = '0' ) ) then
        if ( ( STATE = 0 ) and ( CLOCK'event and
            CLOCK = '1' ) ) then

            ----- find out actual length -----
            N := TABLE1( LEN ) ;

            F := TABLE2( TABLE1( LEN ) ) ;

            ----- do state 0 -----
            STAGE_CNT <= 0 ;
            COE_BUF := "00000000" ;
            PTR := ISTO ;
            FFT_CMP <= '1' ;
            STATE <= 1 ;

        elsif ( ( STATE = 1 ) and ( CLOCK'event and
            CLOCK = '1' ) ) then

            ----- do state 1 which is initialization state -----

            IN_E <= '0' ;
            OUT_E <= '0' ;
            R_CNT := 0 ;
            W_CNT := 0 ;
            EN <= '0' ;

            if( ( IN_R = '1' ) ) then
                -- gen. next addr
                STATE <= 3 ;
            else
                STATE <= 7 ;
            end if ;
        end if ;
    end if ;
end process ;

```

```

elseif( (2 <= STATE) and (STATE <= 4 ) )
then
    ---- do state 2, 3, or 4

    ----- read -----

    if( (IN_R = '1' and R_CNT < 2*N and
        CLOCK'event ) ) then

        if( PTR = '0' )then
            --- when RAM_0 is read ----

                if (( CLOCK = '0')) then

                    ADDR_0 <= RD_ADDR_0 ;
                    TRIG_RD_0 <= not( TRIG_RD_0 );

            --- generate next addr ----
                    ADDR_WC <= COE_BUF ;
                    CHS_WC <= '1',
                                '0' after 1 ns ,
                                '1' after chs_setup_t ;

                    RW_WC <= '1' ;
                    COE_BUF := INC( COE_BUF ) ;
                    elsif ( CLOCK = '1')then
                        ADDR_0 <= RD_ADDR_1 ;
                        TRIG_RD_1 <= not( TRIG_RD_1 );

            -- generate next addr -----
                    end if ;
                    CHS_0 <= '1',
                                '0' after 1 ns ,
                                '1' after chs_setup_t ;
                    RW_0 <= '1' ;

                elsif( PTR = '1' )then
                    -- when RAM_1 is read ---

                        if (( CLOCK = '0')) then

                            ADDR_1 <= RD_ADDR_0 ;
                            TRIG_RD_0 <= not( TRIG_RD_0 );

            -- generate next addr ---
                            ADDR_WC <= COE_BUF ;
                            CHS_WC <= '1',

```

```

        '0' after 1 ns ,
        '1' after chs_setup_t ;

        RW_WC <= '1' ;
        COE_BUF := INC( COE_BUF ) ;
    elsif ( CLOCK = '1') then
        ADDR_1 <= RD_ADDR_1 ;
        TRIG_RD_1 <= not( TRIG_RD_1 ) ;

-- generate next addr ----
    end if ;
    CHS_1 <= '1',
        '0' after 1 ns ,
        '1' after chs_setup_t ;
    RW_1 <= '1' ;

    end if ;

    R_CNT := R_CNT + 1 ;
    STATE <= 3 ;
    TRIG <= not(TRIG) after del_t;

    elsif( R_CNT = 2*N ) then
        IN_E <= '1' ;
        EN <= '1' ;
    end if ;

----- writing -----
    if( ( OUT_A = '1' and W_CNT < 2*N
        and CLOCK'event )
    or (OUT_A'event and OUT_A = '1')) then

        if( PTR = '0' ) then
            if( CLOCK = '0') then
                ADDR_1 <= WR_ADDR_0 ;
                TRIG_WR_0 <= not( TRIG_WR_0 ) ;
            elsif( CLOCK = '1' ) then
                ADDR_1 <= WR_ADDR_1 ;
                TRIG_WR_1 <= not( TRIG_WR_1 ) ;
            end if ;
            CHS_1 <= '1',
                '0' after 30 ns ,
                '1' after chs_setup_t ;

            RW_1 <= '1',

```

```

        '0' after 30 ns,
        '1' after wrt_setup_t ;

    elsif( PTR = '1') then
        if( CLOCK = '0') then
            ADDR_0 <= WR_ADDR_0 ;
            TRIG_WR_0 <= not( TRIG_WR_0 ) ;

            elsif( CLOCK = '1' ) then
                ADDR_0 <= WR_ADDR_1 ;
                TRIG_WR_1 <= not( TRIG_WR_1 ) ;
            end if ;
            CHS_0 <= '1',
                '0' after 30 ns ,
                '1' after chs_setup_t ;
            RW_0 <= '1',
                '0' after 30 ns,
                '1' after wrt_setup_t ;

        end if ;

        if( CLOCK = '0') then
            S1 <= '0' ;
            S0 <= '1' ;
        elsif( CLOCK = '1') then
            S1 <= '1' ;
            S0 <= '0' ;
        end if ;

        W_CNT := W_CNT + 1 ;
        STATE <= 2 ;

        elsif( W_CNT = 2*N ) then
            OUT_E <= '1' ;
            S1 <= '0' after 500 ns ;
            S0 <= '0' after 500 ns ;
        end if ;

        if((W_CNT = 2*N) and ( R_CNT = 2*N)) then
            STATE <= 7 ;
        end if ;

    ----- do state 7 , increment stage_counter
    elsif ( STATE = 7 ) then
        if ( IN_E = '1' and OUT_E = '1') then
            STAGE_CNT <= STAGE_CNT + 1 ;

```



```

        PTR := NOT( PTR ) ;
        STATE <= 8 ;
    else
        if( IN_E = '1' ) then
            STATE <= 2 ;
        else
            STATE <= 3 ;
        end if ;

        TRIG_RD_0 <= not( TRIG_RD_0 ) ;
        TRIG_RD_1 <= not( TRIG_RD_1 ) ;
        TRIG_WR_0 <= not( TRIG_WR_0 ) ;
        TRIG_WR_1 <= not( TRIG_WR_1 ) ;

    end if ;

    ----- do state 8 which is final -----
    elsif ( STATE = 8 ) then
        if ( STAGE_CNT = (F+1) ) then
            FFT_CMP <= '0' after 500 ns ;
            OSTO <= PTR ;
            STATE <= -1 ;
        elsif( STAGE_CNT <(F+1) ) then
            STATE <= 1 ;
        end if ;

    end if ;

    elsif( CHE = '1' ) then
        IN_E <= '1' ;
        OUT_E <= '1' ;
        S0 <= '0' ;
        S1 <= '0' ;
        OSTO <= '0' ;
        ADDR_0 <= "ZZZZZZZZ";
        CHS_0 <= '1' ;
        RW_0 <= '1' ;
        ADDR_WC <= "ZZZZZZZZ";
        CHS_WC <= '1' ;
        RW_WC <= '1' ;
        ADDR_1 <= "ZZZZZZZZ";
        CHS_1 <= '1' ;
        RW_1 <= '1' ;
        STATE <= 0 ;
    end if ;

end process ;

```

```

process(TRIG_RD_0, TRIG_WR_0, TRIG_RD_1, TRIG_WR_1,
        STAGE_CNT)
    variable jum_dis : INTEGER := 0 ;
    variable addr_dis : INTEGER := 1 ;
    variable i1      : INTEGER := 0 ;
    variable i2      : INTEGER := 0 ;
    variable k1      : INTEGER := 0 ;
    variable k2      : INTEGER := 0 ;
    variable j1      : INTEGER := 0 ;
    variable j2      : INTEGER := 0 ;
    variable L       : INTEGER := 0 ;
begin

    if( STAGE_CNT'event and STAGE_CNT >= 0 ) then
        addr_dis := TABLE1(LEN) / 2**( STAGE_CNT ) ;
        jump_dis := TABLE1(LEN)*2 / 2**( STAGE_CNT ) ;
        i1 := 0 ;
        i2 := 0 ;
        j1 := 0 ;
        j2 := 0 ;
        k1 := 0 ;
        k2 := 0 ;
        L := TABLE1(LEN) ;
    else

        if( STAGE_CNT >= 0 and TRIG_RD_0'event) then
            RD_ADDR_0 <=
                BIT_TO_LOGIC( INT_TO_BITSARRAY(((i1 mod addr_dis) +
                                                    j1*jump_dis),8));

            if( ( (i1+1) mod addr_dis )= 0 ) then
                j1 := j1 + 1 ;
            end if ;
            i1 := i1 + 1;

        end if ;

        if( STAGE_CNT >= 0 and TRIG_RD_1'event) then
            RD_ADDR_1 <=
                BIT_TO_LOGIC( INT_TO_BITSARRAY(((i2 mod addr_dis )
                                                    + addr_dis + j2*jump_dis ) ,8)) ;

            if( ( (i2+1) mod addr_dis )= 0 ) then
                j2 := j2 + 1 ;
            end if ;
            i2 := i2 + 1;

        end if ;

        if( STAGE_CNT >= 0 and TRIG_WR_0'event) then
            WR_ADDR_0<=BIT_TO_LOGIC( INT_TO_BITSARRAY( k1, 8 ) );

```

```

        k1 := k1 + 1 ;
    end if ;

    if( STAGE_CNT >= 0 and TRIG_WR_1'event) then
        WR_ADDR_1 <= BIT_TO_LOGIC( INT_TO_BITSARRAY((k2 +
                                                    L),8));
        k2 := k2 + 1 ;
    end if ;
end if ;

end process ;

```

## APPENDIX G: THE BEHAVIOR OF RAM

```
library fpu, fft;
use fft.basic.all, fpu.refer.all;

----- the size of ram is 256 by 32
entity RAM_256 is
  generic ( read_cycle_t : TIME := 300 ns ;-- read cycle time

           write_cycle_t : TIME := 300 ns ;
                               -- write cycle time

           data_setup_t   : TIME := 150 ns ;
                               -- data setup time

           chs_setup_t    : TIME := 150 ns ;
                               -- chip set up time

           wrt_pulse_width_t : TIME := 150 ns ;
                               -- write pulse width

           chs_access_t :TIME := 50 ns);
                               -- access time from chip select

  port( addr_lines : in LOGIC_ARRAY( 7 downto 0 );
        chs       : in BIT ;      --- it is chip select signal

        rw_en     : in BIT ;
                               --- it is read/write enable
  signal
    i_data_lines : in LOGIC_ARRAY( 31 downto 0 );
    o_data_lines : out LOGIC_ARRAY( 31 downto 0 ));
end RAM_256 ;

library fft,fpu;
use fpu.refer.all, fft.basic.all ;
architecture behavioral of RAM_256 is

  signal addr_buf    : LOGIC_ARRAY( addr_lines'left  downto
                                     addr_lines'right );
```

```

signal i_data_lines_buf : LOGIC_ARRAY( i_data_lines'left
                                         downto i_data_lines'right );
signal rw_en_buf : BIT ;
signal chs_buf : BIT ;

begin

    addr_buf <= addr_lines ;

    i_data_lines_buf <= i_data_lines ;

    rw_en_buf <= rw_en ;

    chs_buf <= chs ;

    ----- when chip is enable -----

    --- check for read cycle timing violation ---
    process(rw_en, chs)
    begin
        if ( (rw_en = '1') and (chs = '0') ) then
            assert addr_buf'delayed( read_cycle_t )'stable
            report " read cycle time error "
            severity error ;
        end if ;
    end process ;

    --- check for write cycle time violation ---
    process(rw_en, chs)
    begin
        if ( rw_en = '0' and chs = '0' ) then
            assert addr_buf'delayed( write_cycle_t )'stable
            report " write cycle time error "
            severity error ;
        end if ;
    end process ;

    --- check for write pules width violation ---
    process(rw_en, chs)
    begin
        if ( rw_en = '0' and chs = '0' ) then

```

```

assert rw_en_buf'delayed( wrt_pulse_width_t )'stable
    report " read/write time error "
    severity error ;
end if ;
end process ;

--- check for chip select setup time violation ---
process(rw_en, chs)
begin
    if ( rw_en = '0' and chs = '0' ) then
        assert chs_buf'delayed( chs_setup_t )'stable
        report " chip select setup time error "
        severity error ;
    end if ;
end process ;

--- check for data setup time violation ---
process(rw_en, chs)
begin
    if ( rw_en = '0' and chs = '0' ) then
        assert i_data_lines_buf'delayed( data_setup_t )'stable
        report " data setup time error "
        severity error ;
    end if ;
end process ;

process(rw_en, chs)
    variable cell_num :INTEGER := 0 ;
    variable data_buf : LOGIC_ARRAY( i_data_lines'left
                                     downto i_data_lines'right );
    variable cell_matrix :
        LOGIC_MATRIX( 0 to (2** addr_lines'length - 1 ) ) ;
begin

cell_num := BITSARRAY_TO_INT( LOGIC_TO_BIT( addr_buf)) ;

---- write mode -----

```

```

    if( (rw_en = '0') and ( chs'event and chs = '0'))
      then
        data_buf := i_data_lines_buf ;
        cell_matrix( cell_num ) := data_buf   ;

---- read mode -----
      elsif((rw_en = '1') and ( chs'event and chs = '0' ) )
      then
        o_data_lines <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ",
          cell_matrix( cell_num ) after chs_access_t ;

---- chip disable -----
      else
        o_data_lines <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      end if ;
end process ;

end behavioral;
```

## APPENDIX H: THE SOURCE FILE OF THE FFT SYSTEM

```

library fpu, fft;
use fpu.refer.all, fpu.read1_file.all, fft.basic.all;
use fft.ram_256, fft.convert.all ;
entity sys2 is
    generic( test_number : POSITIVE := 2 ) ;
        --- form 1 to 6 ---
end ;

```

```

library fpu, fft;
use fpu.refer.all, fpu.read1_file.all, fft.basic.all;
use fft.ram_256, fft.convert.all ;
architecture simple of sys2 is

```

```

    function RESOLVE( bits_1, bits_2: LOGIC_ARRAY)
        return LOGIC_ARRAY is
            variable result: LOGIC_ARRAY( bits_1'left downto
                bits_1'right);
            variable test1 : BOOLEAN ;
            variable test2 : BOOLEAN ;
            begin
                test1 := IS_HI_Z_OR_X( bits_1 ) ;
                test2 := IS_HI_Z_OR_X( bits_2 ) ;
                if( test1 and test2 ) then
                    for i in bits_1'range loop
                        result(i) := 'X' ;
                    end loop ;
                elsif( test1 ) then
                    result := bits_2 ;
                elsif( test2 ) then
                    result := bits_1 ;
                else
                    assert( test1 and test2 )
                    report " bus can not resolve any one input signal "
                        severity error ;
                end if ;
                return result ;
            end RESOLVE ;

```

```

function TABLE1( bits: BIT_ARRAY) return INTEGER is
    variable result :integer := 0 ;
    begin
        result := 2** ( BITSARRAY_TO_INT( bits)+ 1) ;
        return result ;
    end

```



```

end TABLE1 ;

function TABLE2( N: INTEGER) return INTEGER is
variable result :integer := 0 ;
begin

    while 2**( result) < N loop
        result := result + 1 ;
    end loop ;
    return result ;

end TABLE2 ;

type vector_set is array( positive range <> ) of
    BIT_ARRAY(2 downto 0) ;

function input_vector return vector_set is
begin
    return( "000",
            "001",
            "010",
            "011",
            "100",
            "100" ) ;
end input_vector ;

component RAM_256
generic( read_cycle_t      : TIME := 300 ns ;
         -- read cycle time
         write_cycle_t     : TIME := 300 ns ;
         -- write cycle time
         data_setup_t      : TIME := 150 ns ;
         -- data setup time
         chs_setup_t       : TIME := 150 ns ;
         -- chip set up time
         wrt_pulse_width_t : TIME := 150 ns;
         -- write pulse width
         chs_access_t      : TIME := 50 ns);
         -- access time from chip select

port( addr_lines : in LOGIC_ARRAY( 7 downto 0 );
      chs        : in BIT ;
      --- active low chip select signal
      rw_en      : in BIT ;
      --- active low write/read enable signal
      i_data_lines : in LOGIC_ARRAY( 31 downto 0 );

```

```

        o_data_lines : out LOGIC_ARRAY( 31 downto 0 ));

end component ;

component FFT_CELL
generic ( D_FPU_T : TIME := 110 ns );
port( a_real,a_img : in LOGIC_ARRAY( 31 downto 0);
      -- a is the input signal.
      b_real,b_img : in LOGIC_ARRAY( 31 downto 0);
      -- b is the input signal.
      w_real,w_img : in LOGIC_ARRAY( 31 downto 0);
      -- w is the weight signal.
      clock : in BIT := '1' ;
      enable : in BOOLEAN := false ;
      -- chip enable for am29325
      ie      : in BOOLEAN := FALSE ;
      -- input enable for final stage output
      oe      : in BOOLEAN := FALSE ;
      -- output enable for first stage input
      c_real,c_img : out LOGIC_ARRAY( 31 downto 0) ;
      -- c is the output signal.
      d_real,d_img : out LOGIC_ARRAY( 31 downto 0));
      -- d is the output signal.

end component ;

for F1:FFT_CELL use entity fft.FFT_CELL( structural );

for all :RAM_256 use entity fft.RAM_256( behavioral );

constant del_t      : TIME := 100 ns ;
constant chs_setup_t : TIME := 200 ns ;
constant wrt_setup_t : TIME := 200 ns ;

signal LEN          : BIT_ARRAY( 2 DOWNT0 0 ) := "000" ;

signal ISTO          : BIT := '1' ;
signal CHE            : BIT := '1' ;
signal IN_R           : BIT := '0' ;
signal OUT_A          : BIT := '0' ;

signal IN_E           : BIT := '1' ;
signal OUT_E          : BIT := '1' ;
signal FFT_CMP        : BIT := '0' ;
signal STAGE_CNT      : INTEGER := -1 ;
signal OSTO           : BIT := '1' ;
signal TRIG           : BIT := '0' ;
signal EN             : BIT := '1' ;
signal SC             : BIT := '0' ;
signal S1             : BIT := '0' ;

```

```

signal      ADDR_0      : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      CHS_0       : BIT := '1'      ;
signal      RW_0        : BIT := '1'      ;
signal      ADDR_WC     : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      CHS_WC      : BIT := '1'      ;
signal      RW_WC       : BIT := '1'      ;
signal      ADDR_1      : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      CHS_1       : BIT := '1'      ;
signal      RW_1        : BIT := '1'      ;

signal      TRIG_RD_0   : BIT      := '0' ;
signal      TRIG_WR_0   : BIT      := '0' ;
signal      TRIG_RD_1   : BIT      := '0' ;
signal      TRIG_WR_1   : BIT      := '0' ;
signal      RD_ADDR_0   : LOGIC_ARRAY( 7 DOWNTO 0)
                                := "ZZZZZZZZ" ;
signal      RD_ADDR_1   : LOGIC_ARRAY( 7 DOWNTO 0)
                                := "ZZZZZZZZ" ;
signal      WR_ADDR_0   : LOGIC_ARRAY( 7 DOWNTO 0)
                                := "ZZZZZZZZ" ;
signal      WR_ADDR_1   : LOGIC_ARRAY( 7 DOWNTO 0)
                                := "ZZZZZZZZ" ;

signal      CLOCK       : BIT := '1'      ;
signal      times       : integer := 0 ;

signal      IE          : BOOLEAN := FALSE ;
signal      OE          : BOOLEAN := FALSE ;
signal      ENABLE      : BOOLEAN := FALSE ;
signal      STATE       : INTEGER := 0 ;

signal      EADDR_0     : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      ECHS_0      : BIT := '1'      ;
signal      ERW_0       : BIT := '1'      ;
signal      EADDR_WC    : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      ECHS_WC     : BIT := '1'      ;
signal      ERW_WC      : BIT := '1'      ;
signal      EADDR_1     : LOGIC_ARRAY( 7 downto 0)
                                := "ZZZZZZZZ";
signal      ECHS_1      : BIT := '1'      ;
signal      ERW_1       : BIT := '1'      ;
signal      S2          : BIT := '0'      ;

signal      CH_0        : BIT := '1'      ;
signal      CH_1        : BIT := '1'      ;

```



```

        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
signal    out2_real      : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

signal    ex_img         : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
signal    ex_real        : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

signal    exW_real       : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
signal    exW_img        : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

signal    FFT_img        : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
signal    FFT_real       : LOGIC_ARRAY(31 downto 0)
        := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
signal    DONE           : BOOLEAN
        := false ;
signal    F              : INTEGER
        := 0 ;
signal    N              : INTEGER
        := 0 ;
signal    L              : BIT_ARRAY( 2 downto 0)
        := "000" ;

begin

    CLOCK <= NOT( CLOCK ) after 500 ns ;

    times <= times + 1 after 1000 ns ;

    assert not( DONE)
        report "this is enough -- good"
        severity error ;

-----
----- resolved signal -----
-----

CH_0 <= CHS_0 and ECHS_0 ;    ----- active low
CH_1 <= CHS_1 and ECHS_1 ;    ----- active low
CH_W <= CHS_WC and ECHS_WC ;  ----- active low
RW_EN_0 <= RW_0 and ERW_0 ;   ----- active low
RW_EN_1 <= RW_1 and ERW_1 ;   ----- active low
RW_EN_W <= RW_WC and ERW_WC ; ----- active low
ADDR_LINES_0 <= RESOLVE( ADDR_0 , EADDR_0 ) ;
ADDR_LINES_1 <= RESOLVE( ADDR_1 , EADDR_1 ) ;

```



```

elsif ( times <= N and (CLOCK'event)) then
    EADDR_0 <= INC( EADDR_0 ) ;
    EADDR_WC <= INC( EADDR_WC ) ;
elsif( times <= (N*(F+1)/2) and (CLOCK'event)) then
    EADDR_WC <= INC( EADDR_WC ) ;
elsif( times = (N*(F+1)/2+1) ) then
    EADDR_0 <= "ZZZZZZZZ" ;
    EADDR_WC <= "ZZZZZZZZ" ;
    S2 <= '0' ;
end if ;

```

```

if( times <= N and ( CLOCK'event ) ) then

```

```

    ECHS_0 <= '1',
              '0' after 1 ns ,
              '1' after chs_setup_t ;
    ERW_0 <= '1',
              '0' after 1 ns ,
              '1' after wrt_setup_t ;

```

```

    ECHS_WC <= '1',
               '0' after 1 ns ,
               '1' after chs_setup_t ;
    ERW_WC <= '1',
               '0' after 1 ns ,
               '1' after wrt_setup_t ;

```

```

elsif ( times <= (N*(F+1)/2) and ( CLOCK'event ))
then

```

```

    ECHS_WC <= '1',
               '0' after 1 ns ,
               '1' after chs_setup_t ;
    ERW_WC <= '1',
               '0' after 1 ns ,
               '1' after wrt_setup_t ;

```

```

else

```

```

    ECHS_WC <= '1';
    ERW_WC <= '1';

```

```

end if ;

```

```

if ( times < (N*(F+1)/2+1) ) then
    CHE <= '1' ;
elsif( times = (N*(F+1)/2+1) ) then
    CHE <= '1', '0' after 10 ns ;
    LEN <= L;
    ISTO <= '0';
end if ;

```

-----end of program -----

```

    if ( (FFT_CMP = '0' and FFT_CMP'event)
        and (times >= 1) ) then
        CHE <= '1' ;
        DONE <= TRUE ;
    end if ;
end process ;

```

```

-----
----- FFT controller -----
process( CLOCK, IN_E, OUT_E )
variable CNT : INTEGER := 0 ;
begin
    if ( (IN_E='0' and IN_E'event ) and
        ( OUT_E='0'and OUT_E'event ) ) then
        CNT := 0 ;
        IN_R <= '1' ;
        OUT_A <='0' ;
        IE <= TRUE ;
        ENABLE <= TRUE ;
    elsif(( CLOCK'event and CLOCK = '0')) then
        CNT := CNT + 1 ;
        if( CNT = 4 ) then
            OE <= TRUE ;
        elsif( CNT = 5 ) then
            OUT_A <= '1' ;
        end if ;
    elsif( (CNT >=4) and (OUT_E = '1') and (CLOCK'event))
    then
        OUT_A <= '0' ;
        ENABLE <= FALSE ;
        OE <= FALSE after 500 ns ;
    elsif( (CNT >=4) and ( IN_E ='1') ) then
        IN_R <= '0' ;
        IE <= FALSE ;
    end if ;
end process ;

```

```

-----
--
-----address sequencer -----
----- generate step by step signal -----
process( CLOCK, LEN, CHE, STATE, IN_R, OUT_A )
variable R_CNT      : INTEGER := 0 ;
variable W_CNT      : INTEGER := 0 ;
variable PTR        : BIT      := '0' ;
variable COE_BUF    : LOGIC_ARRAY( 7 downto 0)
                    := "00000000" ;

```



```

begin
  if ( ( CHE = '0' ) ) then
    if ( ( STATE = 0 ) and ( CLOCK'event
      and CLOCK = '1' ) ) then

      ----- find out actual length -----

      ---- do state 0 -----
      STAGE_CNT <= 0 ;
      COE_BUF := "00000000" ;
      PTR := ISTO ;
      FFT_CMP <= '1' ;
      STATE <= 1 ;

      elsif ( ( STATE = 1 ) and
        ( CLOCK'event and CLOCK = '1' ) ) then

        ----- do state 1 which is initialization state -----

          IN_E <= '0' ;
          OUT_E <= '0' ;
          R_CNT := 0 ;
          W_CNT := 0 ;
          EN <= '0' ;

          if( ( IN_R = '1' ) ) then
            -- gen. next addr
            STATE <= 3 ;
          else
            STATE <= 7 ;
          end if ;

          elsif( ( 2 <= STATE ) and ( STATE <= 4 ) )
            then

              ---- do state 2, 3, or 4

              ----- read -----

                if( ( IN_R = '1' and R_CNT < 2*N and
                  CLOCK'event ) ) then

                  if( PTR = '0' ) then
                    --- when RAM_0 is read

```

```

if (( CLOCK = '0')) then

    ADDR_0 <= RD_ADDR_0 ;
    TRIG_RD_0 <= not( TRIG_RD_0 );
    --generate next addr

    ADDR_WC <= COE_BUF ;
    CHS_WC <= '1',
        '0' after 1 ns ,
        '1' after chs_setup_t ;

    RW_WC <= '1' ;
    COE_BUF := INC( COE_BUF ) ;
    elsif ( CLOCK = '1')then
        ADDR_0 <= RD_ADDR_1 ;
        TRIG_RD_1 <= not( TRIG_RD_1 );
        --generat next addr

    end if ;
    CHS_0 <= '1',
        '0' after 1 ns ,
        '1' after chs_setup_t ;
    RW_0 <= '1' ;

elsif( PTR = '1' )then
    -- when RAM_1 is read

    if (( CLOCK = '0')) then

        ADDR_1 <= RD_ADDR_0 ;
        TRIG_RD_0 <= not( TRIG_RD_0 );
        --generate next addr

        ADDR_WC <= COE_BUF ;
        CHS_WC <= '1',
            '0' after 1 ns ,
            '1' after chs_setup_t ;

        RW_WC <= '1' ;
        COE_BUF := INC( COE_BUF ) ;
        elsif ( CLOCK = '1')then
            ADDR_1 <= RD_ADDR_1 ;
            TRIG_RD_1 <= not( TRIG_RD_1 );
            --generate next addr
        end if ;
        CHS_1 <= '1',
            '0' after 1 ns ,
            '1' after chs_setup_t ;
        RW_1 <= '1' ;

    end if ;

```

```

R_CNT    := R_CNT + 1 ;
STATE <= 3 ;
TRIG <= not(TRIG) after del_t;

elsif( R_CNT = 2*N ) then
    IN_E <= '1' ;
    EN <= '1' ;
end if ;

```

----- writing -----

```

if( ( OUT_A = '1' and W_CNT < 2*N and
CLOCK'event ) or (OUT_A'event and OUT_A = '1'))
then

```

```

    if( PTR = '0' ) then
        if( CLOCK = '0') then
            ADDR_1 <= WR_ADDR_0 ;
            TRIG_WR_0 <= not( TRIG_WR_0 ) ;
            elsif( CLOCK = '1' ) then
                ADDR_1 <= WR_ADDR_1 ;
                TRIG_WR_1 <= not( TRIG_WR_1 ) ;
            end if ;
            CHS_1 <= '1',
                    '0' after 30 ns ,
                    '1' after chs_setup_t ;

            RW_1 <= '1',
                    '0' after 30 ns,
                    '1' after wrt_setup_t ;

```

```

    elsif( PTR = '1') then
        if( CLOCK = '0') then
            ADDR_0 <= WR_ADDR_0 ;
            TRIG_WR_0 <= not( TRIG_WR_0 ) ;

            elsif( CLOCK = '1' ) then
                ADDR_0 <= WR_ADDR_1 ;
                TRIG_WR_1 <= not( TRIG_WR_1 ) ;
            end if ;
            CHS_0 <= '1',
                    '0' after 30 ns ,
                    '1' after chs_setup_t ;
            RW_0 <= '1',
                    '0' after 30 ns,
                    '1' after wrt_setup_t ;

```

```

end if ;

```

```

    if( CLOCK = '0') then
        S1 <= '0' ;
        S0 <= '1' ;
    elsif( CLOCK = '1') then
        S1 <= '1' ;
        S0 <= '0' ;
    end if ;

    W_CNT := W_CNT + 1 ;
    STATE <= 2 ;

    elsif( W_CNT = 2*N ) then
        OUT_E <= '1' ;
        S1 <= '0' after 500 ns ;
        S0 <= '0' after 500 ns ;
    end if ;

    if((W_CNT = 2*N) and ( R_CNT = 2*N))
    then
        STATE <= 7 ;
    end if ;

```

----- do state 7 , increment stage\_counter

```

    elsif ( STATE = 7 ) then
        if ( IN_E = '1' and OUT_E = '1') then
            STAGE_CNT <= STAGE_CNT + 1 ;
            PTR := NOT( PTR ) ;
            STATE <= 8 ;
        else
            if( IN_E = '1' ) then
                STATE <= 2 ;
            else
                STATE <= 3 ;
            end if ;

            TRIG_RD_0 <= not( TRIG_RD_0 ) ;
            TRIG_RD_1 <= not( TRIG_RD_1 ) ;
            TRIG_WR_0 <= not( TRIG_WR_0 ) ;
            TRIG_WR_1 <= not( TRIG_WR_1 ) ;

        end if ;

```

----- do state 8 which is final -----  
 elsif ( STATE = 8 ) then

```

        if ( STAGE_CNT = (F+1) ) then
            FFT_CMP <= '0' after 500 ns ;
            OSTO <= PTR ;
            STATE <= -1 ;
        elsif( STAGE_CNT <(F+1) ) then
            STATE <= 1 ;
        end if ;

    end if ;

    elsif( CHE = '1' ) then
        IN_E <= '1' ;
        OUT_E <= '1' ;
        SO <= '0' ;
        S1 <= '0' ;
        OSTO <= '0' ;
        ADDR_0 <= "ZZZZZZZZ";
        CHS_0 <= '1' ;
        RW_0 <= '1' ;
        ADDR_WC <= "ZZZZZZZZ";
        CHS_WC <= '1' ;
        RW_WC <= '1' ;
        ADDR_1 <= "ZZZZZZZZ";
        CHS_1 <= '1' ;
        RW_1 <= '1' ;
        STATE <= 0 ;
    end if ;

end process ;

process(TRIG_RD_0, TRIG_WR_0, TRIG_RD_1,
        TRIG_WR_1, STAGE_CNT)
    variable jump_dis : INTEGER := 0 ;
    variable addr_dis : INTEGER := 1 ;
    variable i1       : INTEGER := 0 ;
    variable i2       : INTEGER := 0 ;
    variable k1       : INTEGER := 0 ;
    variable K2       : INTEGER := 0 ;
    variable j1       : INTEGER := 0 ;
    variable j2       : INTEGER := 0 ;
    variable L        : INTEGER := 0 ;
begin

    if( STAGE_CNT'event and STAGE_CNT >= 0 ) then
        addr_dis := TABLE1(LEN) / 2**( STAGE_CNT ) ;
        jump_dis := TABLE1(LEN)*2 / 2**( STAGE_CNT) ;
        i1 := 0 ;
        i2 := 0 ;
    end if ;
end process ;

```

```

    j1 := 0 ;
    j2 := 0 ;
    k1 := 0 ;
    k2 := 0 ;
else
    if( STAGE_CNT >= 0 and TRIG_RD_0'event) then
        RD_ADDR_0 <=
            BIT_TO_LOGIC( INT_TO_BITSARRAY(((i1 mod addr_dis) +
                                                j1*jump_dis),8));

        if( ( (i1+1) mod addr_dis )= 0 ) then
            j1 := j1 + 1 ;
        end if ;
        i1 := i1 + 1;

    end if ;

    if( STAGE_CNT >= 0 and TRIG_RD_1'event) then
        RD_ADDR_1 <=
            BIT_TO_LOGIC( INT_TO_BITSARRAY(((i2 mod addr_dis )
                                                + addr_dis + j2*jump_dis ) ,8)) ;

        if( ( (i2+1) mod addr_dis )= 0 ) then
            j2 := j2 + 1 ;
        end if ;
        i2 := i2 + 1;

    end if ;

    if( STAGE_CNT >= 0 and TRIG_WR_0'event) then
        WR_ADDR_0 <= BIT_TO_LOGIC(INT_TO_BITSARRAY( k1, 8));
        k1 := k1 + 1 ;
    end if ;

    if( STAGE_CNT >= 0 and TRIG_WR_1'event) then
        WR_ADDR_1 <= BIT_TO_LOGIC(INT_TO_BITSARRAY((k2+N),8));
        k2 := k2 + 1 ;
    end if ;
end if ;

end process ;

----- simply depict the behavioral of 4 to 1 switch --

process( out1_real, out1_img, out2_real, out2_img,
         ex_real, ex_img, S0, S1, S2 )
variable test : BIT_ARRAY( 2 downto 0 ) := "000" ;
begin
    test := S0&S1&S2 ;

```



```

R0_r:RAM_256 generic map( read_cycle_t      => 300 ns ,
                           write_cycle_t     => 300 ns ,
                           data_setup_t      => 150 ns ,
                           chs_setup_t       => 150 ns ,
                           wrt_pulse_width_t => 150 ns ,
                           chs_access_t      => 50  ns )
port map(ADDR_LINES_0, CH_0, RW_EN_0,R_in_real,R0_real);

R0_i:RAM_256 generic map( read_cycle_t      => 300 ns ,
                           write_cycle_t     => 300 ns ,
                           data_setup_t      => 150 ns ,
                           chs_setup_t       => 150 ns ,
                           wrt_pulse_width_t => 150 ns ,
                           chs_access_t      => 50  ns )
port map( ADDR_LINES_0, CH_0, RW_EN_0, R_in_img, R0_img );

R1_r:RAM_256 generic map( read_cycle_t      => 300 ns ,
                           write_cycle_t     => 300 ns ,
                           data_setup_t      => 150 ns ,
                           chs_setup_t       => 150 ns ,
                           wrt_pulse_width_t => 150 ns ,
                           chs_access_t      => 50  ns )
port map( ADDR_LINES_1, CH_1, RW_EN_1, R_in_real, R1_real);

R1_i:RAM_256 generic map( read_cycle_t      => 300 ns ,
                           write_cycle_t     => 300 ns ,
                           data_setup_t      => 150 ns ,
                           chs_setup_t       => 150 ns ,
                           wrt_pulse_width_t => 150 ns ,
                           chs_access_t      => 50  ns )
port map( ADDR_LINES_1, CH_1, RW_EN_1, R_in_img,
          R1_img );

W_r:RAM_256 generic map( read_cycle_t      => 300 ns ,
                           write_cycle_t     => 300 ns ,
                           data_setup_t      => 150 ns ,
                           chs_setup_t       => 150 ns ,
                           wrt_pulse_width_t => 150 ns ,
                           chs_access_t      => 50  ns )
port map( ADDR_LINES_W, CH_W, RW_EN_W, exW_real,
          W_real );

W_i:RAM_256 generic map( read_cycle_t      => 300 ns ,

```



```

write_cycle_t      => 300 ns ,
data_setup_t       => 150 ns ,
chs_setup_t        => 150 ns ,
wrt_pulse_width_t  => 150 ns ,
chs_access_t       => 50  ns )
port map( ADDR_LINES_W, CH_W, RW_EN_W, exW_img,
          W_img );

```

```

end simple;

```

## APPENDIX I: THE ACCESSORY FILES

### A. THE SOURCE FILE ASSOCIATED WITH DATA READ

```
library fpu;
use STD.TEXTIO.all;
package READ1_FILE is

    type REAL_MATRIX is array( integer range <> ) of real ;

    procedure read_real(F_name:in STRING ;
                        data_array:out REAL_MATRIX);

end READ1_FILE ;

library fpu;
use STD.TEXTIO.all;
package body READ1_FILE is

    procedure read_real(F_name:in string; data_array: out
                        REAL_MATRIX) is

        --- this procedure is design for input real data

        file F: text is in F_name;
        variable temp: LINE;
        variable temp_data:real;
        variable L_flag: BOOLEAN := true;
        variable count : INTEGER := 1;
        begin

            -- extract the real data_array from data file.
            while ( not endfile(F)) loop
                readline(F, temp) ;
                read(temp,temp_data);
                data_array(count):= temp_data ;
                count := count + 1 ;
            end loop;
        end read_real;

end READ1_FILE;
```

```

library fpu;
use STD.TEXTIO.all,fpu.refer.all;
package READ_FILE is
    function bit_type ( char : CHARACTER )
        return BIT ;
    procedure read_data(F_name:in STRING ; data_array:out
BIT_MATRIX);
end READ_FILE ;

```

```

library fpu;
use STD.TEXTIO.all,fpu.refer.all;
package body READ_FILE is

```

```

    function bit_type( char : CHARACTER)
        return BIT is
        variable b: BIT ;
        begin
            if ( char = '1') then
                b := '1';
            elsif ( char = '0') then
                b := '0';
            end if ;
            return b;
        end bit_type ;

```

```

    procedure read_data(F_name:in string;
        data_array:out BIT_MATRIX) is

```

--- this procedure is design for input data length 32 bits

```

    file F: text is in F_name;
    variable temp: LINE;
    variable temp_char:CHARACTER;
    variable IO_temp: BIT_ARRAY(1 to 32);
    variable L_flag: BOOLEAN := true;
        variable count : INTEGER := 1;
        variable i :integer := 2;
    begin

```

-- cut out the unwanted space or portion.

```

    while not endfile(F) loop
        L_flag := true ;
        i := 2 ;
        readline(F,temp);
        while L_flag loop
            read(temp,temp_char);
            if(temp_char = '1' or temp_char = '0') then
                L_flag := false ;
            end if;

```

```

        end loop ;

-- extract the bits array from data file.
    IO_temp(1) := BIT_TYPE(temp_char) ;
    while (i <= 32) loop
        read(temp,temp_char);
        if( temp_char = '1' or temp_char = '0')
        then
            IO_temp(i) := BIT_TYPE(temp_char) ;
        elsif( endfile(F) ) then
            assert not (temp_char /= '1' and temp_char /= '0')
            report " reach down to the end of data_file. ";
            end if ;
            i := i + 1;
        end loop ;

        data_array(count):= IO_temp ;
        count := count + 1 ;
    end loop;
end read_data;
end READ_FILE;

```

#### B. THE SOURCE FILE OF THE CONVERSION BETWEEN FP\_NUMBER AND IEEE FORMAT

```

library fpu ;
use fpu.refer.all;
package CONVERT is

    function CONVERT1( value : REAL )
        return BIT_ARRAY ;

end CONVERT ;

package body CONVERT is

    --- convert fp_number into IEEE standard format -----
    ----- procession = 32
    -----

    function CONVERT1( value : REAL )
        return BIT_ARRAY is
            variable result          : BIT_ARRAY( 31 downto 0 )
                := "00000000000000000000000000000000" ;

```

```

variable mantissa_bits : BIT_ARRAY( 22 downto 0 ) ;
variable exp_bits      : BIT_ARRAY( 7  downto 0 ) ;
variable sign          : BIT ;
variable quot          : INTEGER := 0 ;
variable local         : REAL := 0.0 ;
begin
  if( value > 0.0) then
    sign := '0' ;
  elsif( value < 0.0) then
    sign := '1' ;
  elsif( value = 0.0 ) then
    return result ;
  end if ;

  local := abs(value) ;

  while (local >= 2.0) or ( local < 1.0) loop
    if ( local >= 2.0 ) then
      local := local * 0.5 ;
      quot := quot + 1 ;
    elsif( local < 1.0 ) then
      local := local * 2.0 ;
      quot := quot - 1 ;
    end if ;
  end loop ;

  mantissa_bits :=
    FP_TO_BITSARRAY( (local-1.0),mantissa_bits'length ) ;

  exp_bits :=
    INT_TO_BITSARRAY( (quot+127), exp_bits'length) ;
  result := sign & exp_bits & mantissa_bits ;

  return result ;

end CONVERT1 ;

end CONVERT ;

```

## LIST OF REFERENCES

1. VHDL MANUAL, 2<sup>nd</sup>ed., IEEE 1.76, 1989.
2. Lipsett, R., Schaefer, C.F., and Ussery, C., VHDL: Hardware Description And Design, Kluwer Academic Publishers, 1989.
3. J. R. Armstrong, CHIP-LEVEL MODELING WITH VHDL, Prentice Hall, 1989.
4. L. H. Pollard, Computer Design And Architecture, Prentice Hall, 1990, page 49.
5. J. L. Heanesy & D.A. Patterson, Computer Architecture & Quantitative Approach, Morgan Kaufmann, page A-14.
6. D. Stevenson, "A Proposed Standard For Binary Floating point Arithmetic", IEEE Computer, March 1981.
7. S. Carlson, Introduction To HDL-Based Design Using VHDL, Synopsys, Inc., page 5.
8. R. D. Strum and D. E. Kirk, First Principles Of Discrete Systems And Digital Signal Processing, Addison wesley, 1988, pages 466 - 522.
9. A. K. Jain, Fundamentals Of Digital Image Processing, Prentice Hall, 1989, pages 150 - 151.
10. Array Processing And Digital Signal Processing Hand Book, pages 18-20.
11. A. J. Kern and T. E. Cutis, "A Fast 32-bits Complex Vector Processing Engine", Proceeding Of The Institute Of Acoustics, Vol 11 part 8, 1989.

# INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5100	1
4. Professor Chin-Hwa Lee, Code EC/Le Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5100	5
5. Professor Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5100	1
6. Y.S. Wu, Code 8120 Naval Research Laboratory Washington, DC, 20375	1
7. Hu, Ta-Hsiang 26 LANE415, LEIN WU RD, TAICHUNG, TAIWAN 40124 R. O. C.	3